Tablice jednowymiarowe w języku Python

- Wprowadzenie
- Przeczytaj
- Prezentacja multimedialna
- Sprawdź się
- Dla nauczyciela

Tablice jednowymiarowe w języku Python

Źródło: Roman Synkevych, domena publiczna.

> W tym e-materiale powtarzamy wiadomości ze szkoły podstawowej.

W otaczającym świecie możemy znaleźć wiele struktur, które są danymi ułożonymi w pewnej kolejności. Tablica jest strukturą danych pozwalającą na przetwarzanie danych tego samego typu. Przykładem jest między innymi kod DNA, czyli sekwencja następujących po sobie elementów.

W języku Python tablica może być zaimplementowana np. za pomocą zmiennych typu `list` (listy to uporządkowane sekwencje wartości różnego typu).

Więcej informacji o tablicach jednowymiarowych znajdziesz w e-materiale: Tablica jednowymiarowa. Ciekawi cię, jak wyglądają implementacje list jednowymiarowych w innych językach programowania? Możesz się z nimi zapoznać w dwóch pozostałych e-materiałach z tej serii:

- Tablice jednowymiarowe w języku C++,
- Tablice jednowymiarowe w języku Java.

Więcej zadań? Sięgnij do Tablice jednowymiarowe – zadania maturalne.

> **Twoje cele**

- Określisz, w jaki sposób w języku Python implementuje się tablice.
- Prześledzisz, jak definiuje się listy oraz krotki.
- Przeanalizujesz algorytm przeszukiwania listy i szukania najdłuższego spójnego niemalejącego podciągu.
- Skonstruujesz algorytm i zaimplementujesz program przeszukujący zestaw danych.
- Scharakteryzujesz metody specyficzne dla struktury `list` w języku Python.
- Wymienisz zastosowania list oraz krotek.

Przeczytaj

Tablica jednowymiarowa – przypomnienie

Tablica jednowymiarowa jest uporządkowanym zbiorem elementów tego samego typu. Są one numerowane (indeksowane). Pierwszy element ma numer 0.

Tablica jednowymiarowa w języku Python

W języku Python struktura danych (tablica jednowymiarowa) implementowana jest najczęściej za pomocą **listy**.

Oto deklaracja i zarazem definicja pustej tablicy jednowymiarowej w języku Python:

```
1 przykladowa_tablica = []
```

Definiowanie tablicy zaczynamy od nadania nazwy. Nie musimy określać, jakiego typu dane będzie przechowywać.

Do nazwy `przykladowa_tablica` będziemy się odwoływać, aby odczytać zawartość tablicy, wpisać konkretną liczbę w określonym miejscu albo w celu wykonania operacji na wybranym elemencie.

Nie musimy w tym miejscu deklarować rozmiaru tablicy.

> **Dla zainteresowanych**
>
> W języku Python istnieje również typ `tuple` (krotka), który jest bardzo podobny do listy, jednak jego modyfikacja nie jest możliwa. Kolejną różnicą pomiędzy nimi jest to, że zmienna typu `tuple` zajmuje mniej pamięci niż `list`.

Operacje na listach

> **Przykład 1**
>
> Listy w języku Python są modyfikowalne. Oto wybrane metody, które możemy zastosować dla obiektu L (typu `list`):

Nazwa metody	Parametr	Zwracana wartość	Opis metody – uwagi

Nazwa metody	Parametr	Zwracana wartość	Opis metody – uwagi
`L.append(x)`	`x` – dowolny typ	-	dopisuje do listy (na jej końcu) przekazany obiekt `x`
`L.copy()`	-	`list`	tworzy nowy obiekt i zwraca duplikat listy
`L.clear()`	-	-	usuwa zawartość całej listy (kasuje wszystkie jej elementy)
`L.count(x)`	`x` – dowolny typ	`int`	zwraca liczbę wystąpień obiektu `x` na liście
`L.extend(L2)`	`L2 - Iterable`	-	rozszerza (dodaje) do listy elementy obiektu iterowanego L2
`L.index(x, start, stop)`	`x` – dowolny typ, `start`, `stop` – opcjonalne; określają przedział dla przeszukiwania	`int`	zwraca index pierwszego elementu w liście o wartości `x` (lub zgłasza wyjątek `ValueError`, jeśli taki obiekt nie występuje)

Przykład 2

Dana jest tablica jednowymiarowa `liczby`. Jest to pusta tablica.

```
liczby = []
```

Wypełnimy ją danymi, wykorzystując pętlę `for`. Wypełnimy ją kolejnymi liczbami naturalnymi z zakresu `<a, b)`.

Zmienna `liczba` przechowuje kolejne wartości z zakresu `<a, b)`.

```
liczby = []
a = 1
b = 10

for liczba in range(a,b):
    liczby.append(liczba)

print(liczby)
```

Przykład 3

W języku Python tablice implementowane są za pomocą listy. Listy od tablic różnią się tym, że mogą zawierać różne typy danych. Zatem poniższa struktura jest listą, ale nie jest tablicą:

```
lista_danych = [27, "Grzegorz", True, "maj", 0.05]
```

Natomiast poniższa struktura jest tablicą zaimplementowaną za pomocą listy:

```
tablica_danych = ["Michał", "Grzegorz", "Karol", "Piotr", "Pawe
```

Zmienna `tablica_danych` składa się z pięciu elementów – do każdego z nich możemy się odwołać, podając indeks, czyli numer określający pozycję elementu.

Przygotujmy kod, który pozwoli wypisać elementy tablicy (`element`), ich indeks (`ind`) oraz typ (`type()`).

Wbudowana funkcja `enumerate` pozwala przeiterować po tablicy i pobrać zarówno wartość indeksu, jak i elementu znajdującego się pod danym indeksem.

Do wyświetlenia danych użyjemy mechanizmu formatowania f-string:

```
for ind, element in enumerate(tablica_danych):
```

```
2     print(f"Element o indeksie {ind} ma wartość {element}")
3     print(f"Jego typ to {type(element)}")
```

Efekt działania programu:

```
 1 Element o indeksie 0 ma wartość Michał
 2 Jego typ to <class 'str'>
 3 Element o indeksie 1 ma wartość Grzegorz
 4 Jego typ to <class 'str'>
 5 Element o indeksie 2 ma wartość Karol
 6 Jego typ to <class 'str'>
 7 Element o indeksie 3 ma wartość Piotr
 8 Jego typ to <class 'str'>
 9 Element o indeksie 4 ma wartość Paweł
10 Jego typ to <class 'str'>
```

Ważne!

- Pozycja obiektu w ciągu elementów pozostaje niezmienna, nie możemy przypisać elementowi tablicy innego indeksu.
- Każdy element tablicy musi być tego samego typu.
- W języku Python indeksowanie elementów tablicy rozpoczyna się od liczby 0, zatem zapis `tablica_danych[3]` wskazuje czwarty element.
- Oprócz tablic implementowanych za pomocą list język Python oferuje także krotki (typ `tuple`), które zachowują się podobnie do list (z pewnymi ograniczeniami).

Wykorzystajmy poznane metody w prostym programie.

Stworzymy najpierw pustą listę, którą wypełnimy naprzemiennie liczbami całkowitymi od 0 do 6 i ich odpowiednikami typu `str`. W celu wypełnienia listy danymi, wykorzystamy instrukcję `for`.

Elementy do listy dodajemy za pomocą metody `append()`. Dodaje ona element na końcu listy.

```
1 lista = []
2 for kolejny in range(7):
3     lista.append(kolejny)
4     print(lista)
5     lista.append(str(kolejny))
6     print(lista)
```

Efektem działania programu jest wyświetlenie zawartości listy składającej się z elementów typu `int` oraz typu `str` (w języku Python lista może zawierać elementy różnych typów):

```
[0]
[0, '0']
[0, '0', 1]
[0, '0', 1, '1']
[0, '0', 1, '1', 2]
[0, '0', 1, '1', 2, '2']
[0, '0', 1, '1', 2, '2', 3]
[0, '0', 1, '1', 2, '2', 3, '3']
[0, '0', 1, '1', 2, '2', 3, '3', 4]
[0, '0', 1, '1', 2, '2', 3, '3', 4, '4']
[0, '0', 1, '1', 2, '2', 3, '3', 4, '4', 5]
[0, '0', 1, '1', 2, '2', 3, '3', 4, '4', 5, '5']
[0, '0', 1, '1', 2, '2', 3, '3', 4, '4', 5, '5', 6]
[0, '0', 1, '1', 2, '2', 3, '3', 4, '4', 5, '5', 6, '6']
```

Wykonamy teraz kilka operacji na poszczególnych elementach:

```
for kolejny in range(14):
    if kolejny % 2 == 0:
        lista[kolejny] = lista[kolejny] * 3
    if kolejny % 2 == 1:
        lista[kolejny] = lista[kolejny] * 2
    print(lista)
```

W rezultacie otrzymamy zmodyfikowaną listę:

```
[0, '0', 1, '1', 2, '2', 3, '3', 4, '4', 5, '5', 6, '6']
[0, '00', 1, '1', 2, '2', 3, '3', 4, '4', 5, '5', 6, '6']
[0, '00', 3, '1', 2, '2', 3, '3', 4, '4', 5, '5', 6, '6']
[0, '00', 3, '11', 2, '2', 3, '3', 4, '4', 5, '5', 6, '6']
[0, '00', 3, '11', 6, '2', 3, '3', 4, '4', 5, '5', 6, '6']
[0, '00', 3, '11', 6, '22', 3, '3', 4, '4', 5, '5', 6, '6']
[0, '00', 3, '11', 6, '22', 9, '3', 4, '4', 5, '5', 6, '6']
[0, '00', 3, '11', 6, '22', 9, '33', 4, '4', 5, '5', 6, '6']
[0, '00', 3, '11', 6, '22', 9, '33', 12, '4', 5, '5', 6, '6']
[0, '00', 3, '11', 6, '22', 9, '33', 12, '44', 5, '5', 6, '6']
```

```
11 [0, '00', 3, '11', 6, '22', 9, '33', 12, '44', 15, '5', 6, '6']
12 [0, '00', 3, '11', 6, '22', 9, '33', 12, '44', 15, '55', 6, '6']
13 [0, '00', 3, '11', 6, '22', 9, '33', 12, '44', 15, '55', 18, '6']
14 [0, '00', 3, '11', 6, '22', 9, '33', 12, '44', 15, '55', 18, '66'
```

Pamiętajmy, że w języku Python działanie `n * ciag_znakow`, gdzie n jest typu `int` i `ciag_znakow` jest typu `str`, zwróci `ciag_znakow` powtórzony n razy.

> **Ważne!**
>
> Do elementów list w języku Python można się również odwoływać za pomocą ujemnych indeksów. Za pomocą `lista[-i]` odwołujemy się do i-tego elementu, licząc od końca listy. Przykładowo, wywołując `lista[-1]`, otrzymamy ostatni element, natomiast w przypadku `lista[-3]` – trzeci element od końca.

Zmiana elementów listy

Ponieważ listy w języku Python są modyfikowalne, możemy dowolne typy danych dodawać do listy lub wpisywać w miejsce istniejących elementów. Oto kilka przykładów takich operacji:

```
1  lista = [] # pusta lista
2  lista.append(22)
3  print(lista)
4  # [22]
5
6  lista.extend([3, 55, 32, 2.6, 'Python'])
7  print(lista)
8  # [22, 3, 55, 32, 2.6, 'Python']
9
10 print(lista[3])
11 # 32
12
13 lista[3] = 'Linux'
14 print(lista)
15 # [22, 3, 55, 'Linux', 2.6, 'Python']
16
17 lista.extend('Linux')
18 print(lista)
19 # [22, 3, 55, 'Linux', 2.6, 'Python', 'L', 'i', 'n', 'u', 'x']
```

W obiektach typu `list` można odwołać się do ich poszczególnych elementów, używając wyrażenia indeksującego. Obowiązuje następująca składnia:

```
wycinek = lista[początek : koniec : krok]
```

gdzie:

- `początek` – indeks elementu, od którego będzie się zaczynał podciąg; domyślnie `początek = 0`,
- `koniec` – indeks elementu, przed którym skończy się podciąg; wycinek z listy zawiera elementy listy o indeksach z przedziału <początek, koniec); domyślnie `koniec = len(lista)`,
- `krok` – pozwala określić, co który element podciągu będzie się zawierał w wycinku; domyślnie `krok = 1`.

Wszystkie parametry muszą być liczbami całkowitymi. Indeksy można pomijać, wtedy zostaną zastosowane domyślne wartości. Oto przykłady poprawnych wycinków:

- `lista[:3]` – trzy pierwsze elementy listy,
- `lista[3:]` – elementy listy z pominięciem pierwszych trzech elementów,
- `lista[3:5]` – elementy listy o indeksach od 3 do 4,
- `lista[4:2:-1]` – elementy listy o indeksach od 3 do 4, ale w odwrotnej kolejności (najpierw element `lista[4]`, następnie `lista[3]`).

> **Ważne!**
>
> Zauważ, że jeżeli `krok` jest liczbą ujemną, to elementy w wycinku zostaną zapisane w odwrotnej kolejności. Szczególnie instrukcja `lista[::-1]` pozwala nam odwrócić listę.

Oto klika przykładowych wywołań z użyciem tej konstrukcji:

```
lista = [6, 6, 12, 19, 19, 5, 15, 2, 9, 19, 7, 18, 10, 16, 19]
print(lista[3:6])
# [19, 19, 5]

print(lista[10:6])
# []

print(lista[10:6:-1])
# [7, 19, 9, 2]

print(lista[:])
# [6, 6, 12, 19, 19, 5, 15, 2, 9, 19, 7, 18, 10, 16, 19]

```

```
14 print(lista[::-1])
15 # [19, 16, 10, 18, 7, 19, 9, 2, 15, 5, 19, 19, 12, 6, 6]
```

Ważne!

W języku Python listy nie mają z góry ustalonej wielkości. Odwołując się do elementu listy spoza zakresu, otrzymamy błąd.

Polecenie 1

Napisz funkcję z jednym parametrem typu `lista`. W wyniku działania funkcji powinny być wypisane wszystkie elementy listy wraz z ich typem, oddzielone znakiem „-”. Każdy element musi być wypisany od nowej linii.

Operacje na krotkach (tuplach)

Krotki mają podobne możliwości do list, jednak w odróżnieniu do list są niezmienne.

Przykład 4

Oto sposób definiowania krotki oraz kilka przykładowych jej użyć:

```
1 przykladowa_krotka = (3, 2.14, "Python", True)
2
3 print(przykladowa_krotka[2])
4 # Python
5
6 print(przykladowa_krotka[2:])
7 # ('Python', True)
8
9 print(type(przykladowa_krotka[2]))
10 # <class 'str'>
11
12 print(type(przykladowa_krotka[2:]))
13 # <class 'tuple'>
```

Podczas operacji wyświetlania nieistniejącego elementu krotki, Python zgłosi błąd. W przypadku próby wpisania danych do istniejącego elementu, również zostanie zgłoszony wyjątek.

```
1 przykladowa_krotka = (3, 2.14, "Python", True)
```

```
2
3 print(type(przykladowa_krotka[15]))
4 # Traceback (most recent call last):
5 #   File "<pyshell#23>", line 1, in <module>
6 #     print(type(przykladowa_krotka[15]))
7 # IndexError: tuple index out of range
8
9 przykladowa_krotka[1] = 2
10 # Traceback (most recent call last):
11 #   File "<pyshell#24>", line 1, in <module>
12 #     przykladowa_krotka[1] = 2
13 # TypeError: 'tuple' object does not support item assignment
```

Możemy wykonywać operację nazywaną **rozpakowywaniem krotek** – polega ona na przypisaniu elementów krotki do pojedynczych zmiennych. Zapoznajmy się z dwoma ciekawymi przykładami.

```
1 przykladowa_krotka = (3, 2.14, "Python", True)
2 a, b, c, d = przykladowa_krotka
3
4 print(a)
5 # 3
6
7 print(b)
8 # 2.14
9
10 print(c)
11 # Python
12
13 print(d)
14 # True
15
16 # ciekawy przykład
17 x = 2
18 y = 5
19 q = -1
20 x, y = q + y, x + y + q
21
22 # ile wynoszą x oraz y?
23 print(x)
24 # 4
```

```
25
26 print(y)
27 # 6
```

Ważne!

Drugi z pokazanych przykładów możemy wykorzystać, jeżeli chcemy zamienić między sobą wartości dwóch zmiennych, nie używając przy tym pomocniczej zmiennej:

Zamiana dwóch zmiennych z użyciem pomocniczej zmiennej:

```
1 a = 1
2 b = 2
3 print(a, b)
4 # 1, 2
5
6 pomocnicza_zmienna = a
7 a = b
8 b = pomocnicza_zmienna
9 print(a, b)
10 # 2, 1
```

Zamiana dwóch zmiennych za pomocą rozpakowywania:

```
1 a = 1
2 b = 2
3 print(a, b)
4 # 1, 2
5
6 a, b = b, a
7 print(a, b)
8 # 2, 1
```

Dla zainteresowanych

Ze względu na sposób, w jaki krotki są przechowywane w pamięci, w języku Python zajmują one mniej miejsca niż listy zawierające takie same dane. Ponadto krotki są szybsze w działaniach typu wyszukiwanie elementów. Należy jednak pamiętać, że dane w nich są niezmienne, dlatego w niektórych sytuacjach ich użycie może okazać się niemożliwe.

Wykorzystanie list w języku Python

Specyfikacja problemu:

Dane:

- `lista_danych` – lista zawierająca liczby rzeczywiste

Wynik:

Program zwraca długość najdłuższego spójnego nierosnącego podciągu w `lista_danych`.

Wyszukiwanie pierwszego najdłuższego spójnego nierosnącego podciągu liczb można łatwo zrealizować, posługując się listami. Dokładny opis algorytmu znajduje się w e-materiale Tablice jednowymiarowe – zadania maturalne. Na początku inicjujemy zmienne `dlugosc_aktualnego_ciagu` oraz `dlugosc_najdluzszego_ciagu`, które będą przechowywać odpowiednio długość obecnego ciągu oraz długość najdłuższego znalezionego nierosnącego ciągu. Ich początkową wartość ustawimy na `1`, ponieważ najkrótszy podciąg nierosnący w niepustym ciągu zawsze ma długość `1`. Następnie wystarczy odczytywać po kolei elementy listy i sprawdzać, czy są one mniejsze od poprzedniego elementu lub czy są mu równe. Jeśli tak, należy zwiększyć długość badanego właśnie nierosnącego podciągu o jeden, w przeciwnym wypadku zmiennej `dlugosc_aktualnego_ciagu` nadajemy wartość 1 – uznajemy, że to początek kolejnego podciągu. Następnie sprawdzamy, czy jego długość jest większa od długości poprzedniego najdłuższego ciągu, jeśli tak – uznajemy ją za maksymalną i aktualizujemy wartość zmiennej `dlugosc_najdłuższego_ciagu`.

Trzeba pamiętać, aby zacząć przeszukiwanie od drugiego elementu listy (o indeksie 1), aby porównać go z poprzednim, czyli pierwszym (o indeksie 0). Zwróćmy uwagę, że długość aktualnego podciągu zawsze zaczyna się od 1. Gdy znajdujemy mniejszy element, to poprzedni (wcześniejszy) już istnieje i należy go wliczyć do podciągu.

Zdefiniujmy funkcję zgodną z podanym algorytmem:

```
1  def badanie_nierosnacego_ciagu(lista_danych):
2      dlugosc_aktualnego_ciagu = 1
3      dlugosc_najdluzszego_ciagu = 1
4  
5      for element in range(1, len(lista_danych)):
6          badany_aktualny = lista_danych[element]
7          badany_poprzedni = lista_danych[element - 1]
8  
9          if badany_aktualny <= badany_poprzedni:
10             dlugosc_aktualnego_ciagu += 1
11         else:
12             dlugosc_aktualnego_ciagu = 1
```

```
13
14         if dlugosc_aktualnego_ciagu > dlugosc_najdluzszego_ciagu:
15             dlugosc_najdluzszego_ciagu = dlugosc_aktualnego_ciagu
16
17     return dlugosc_najdluzszego_ciagu
```

Wyszukajmy długość najdłuższego spójnego i nierosnącego podciągu na przykładowej liście:

```
 1 lista_danych = [789, 499, 746, 50180, 2, 38, 98774,
 2                   46003, 316, 211, 23064, 89783, 197, 474,168,
 3                   54261, 68915, 645, 567, 2187, 949, 93963,
 4                   117, 769, 65518, 8446, 2763, 205, 650, 763,
 5                   73438, 824, 24697, 766, 309, 19019, 989,
 6                   578, 67812, 21963, 424, 193, 16506, 83875,
 7                   382, 51032, 331, 892, 638, 473, 76010,
 8                   15199, 44706, 646, 59049, 132, 1482, 4524,
 9                   7792, 223, 617, 785, 146]
10 print(badanie_nierosnacego_ciagu(lista_danych))
11 # 4
12
13 # są to liczby: 98774, 46003, 316, 211
```

Polecenie 2

Napiszmy program, który wykorzysta przedstawioną funkcję i wypisze, z ilu elementów składa się najdłuższy spójny i nierosnący podciąg liczb na liście.

Elementy ciągu wygenerujemy za pomocą wyrażenia listowego i funkcji `randint()`.

Jeśli użyjemy funkcji `seed(dowolna_liczba)`, wówczas dla ustalonej wartości `dowolna_liczba` zawsze uzyskamy identyczne liczby podczas losowania.

Dla zainteresowanych

Jeśli chcemy dokładnie przeanalizować działanie opisanej funkcji, możemy wzbogacić program o polecenie wyświetlania dodatkowych informacji – np. korzystając z wyrażeń indeksujących, uzyskać dane na temat znajdowanych elementów. Oto zmodyfikowana funkcja i efekt jej działania:

```
1  def badanie_nierosnacego_ciagu(lista):
2      dlugosc_aktualnego_ciagu = 1
3      dlugosc_najdluzszego_ciagu = 1
4  
5      for element in range(1, len(lista)):
6          badany_aktualny = lista[element]
7          badany_poprzedni = lista[element - 1]
8  
9          if badany_aktualny <= badany_poprzedni:
10             dlugosc_aktualnego_ciagu += 1
11         else:
12             dlugosc_aktualnego_ciagu = 1
13 
14         if dlugosc_aktualnego_ciagu > dlugosc_najdluzszego_ciag
15             dlugosc_najdluzszego_ciagu = dlugosc_aktualnego_cia
16             print(dlugosc_najdluzszego_ciagu, '-', lista[elemen
17 
18     return dlugosc_najdluzszego_ciagu
19 
20 # przykładowe wykonanie
21 lista_danych = [789, 499, 449, 746, 50180, 2, 38, 98774, 46003,
22                 316, 211, 23064, 89783, 197, 474, 168,
23                 54261, 68915, 645, 567, 2187, 949, 93963,
24                 117, 769, 65518, 8446, 2763, 205, 650, 763,
25                 73438, 824, 24697, 766, 309, 19019, 989,
26                 578, 67812, 21963, 424, 193, 16506, 83875,
27                 382, 51032, 331, 892, 638, 473, 76010,
28                 15199, 44706, 646, 59049, 132, 1482, 4524,
29                 7792, 223, 617, 785, 146]
30 
31 print(badanie_nierosnacego_ciagu(lista_danych))
32 # 2 - [789, 499]
33 # 3 - [789, 499, 449]
34 # 4 - [98774, 46003, 316, 211]
35 # 4
```

Już wiesz

Podsumujmy najważniejsze elementy tej sekcji:

- oprócz list Python oferuje także krotki (typ `tuple`), które zachowują się podobnie do list, z pewnymi ograniczeniami,

> - w języku Python listy czy krotki nie posiadają z góry ustalonej wielkości,
> - każdy element listy czy krotki może być innego typu,
> - w języku Python indeksowanie rozpoczyna się od zera, zatem zapis `lista_danych[3]` wskazuje czwarty element.

Problem 1

Napisz program, który w podanym ciągu liczb znajdzie pierwszy najdłuższy **niemalejący** spójny podciąg i wypisze go na standardowe wyjście. Rozwiązanie przetestuj dla ciągu `[6,7,1,3,4]`
.

Specyfikacja problemu:

Dane:

- `ciag_liczb` – badany ciąg zawierający liczby całkowite

Wynik:

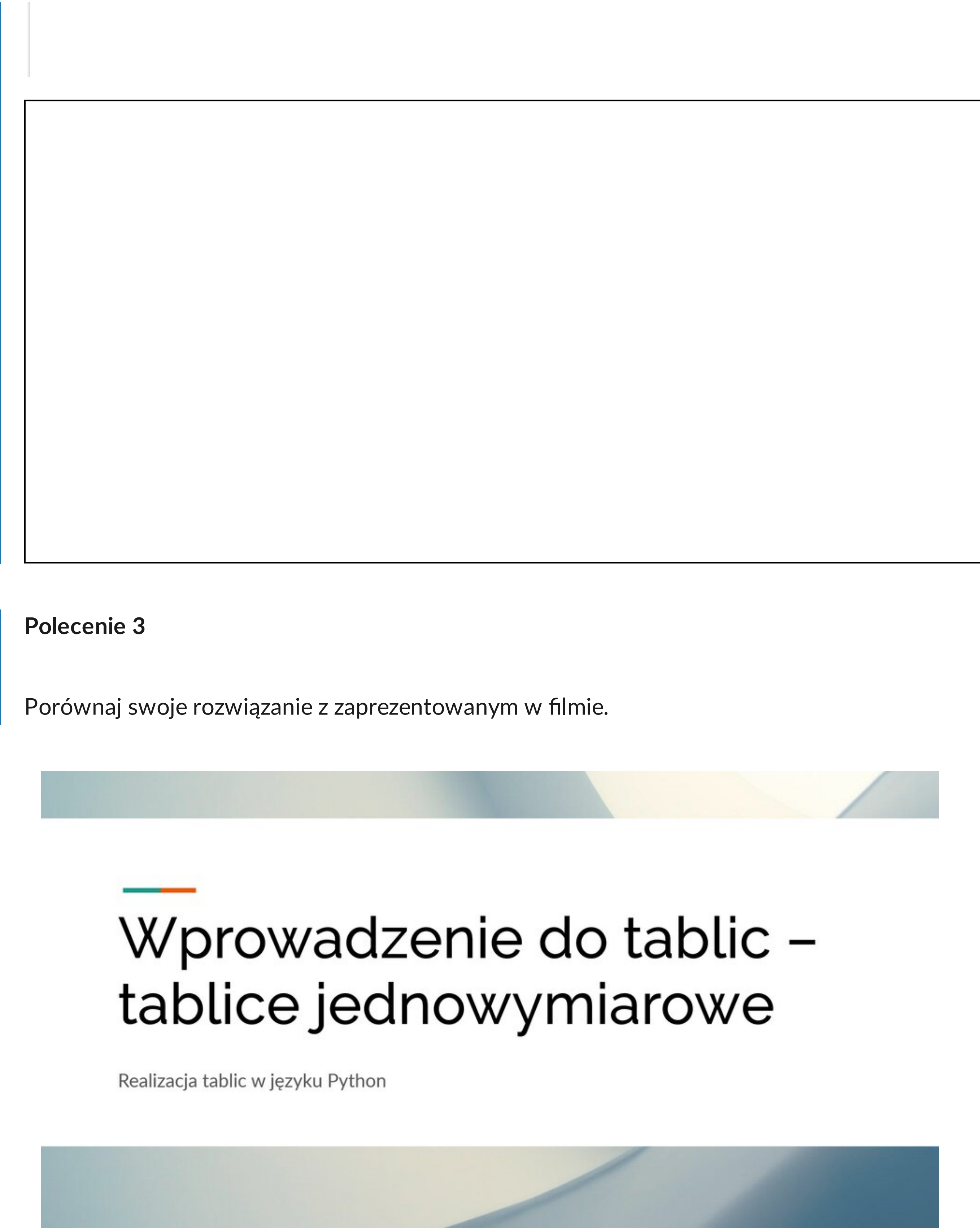
- Program wypisuje na standardowe wyjście znaleziony podciąg.

Twoje zadania

1. Program wypisuje pierwszy najdłuższy niemalejący spójny podciąg.

Polecenie 3

Porównaj swoje rozwiązanie z zaprezentowanym w filmie.



Film dostępny pod adresem /preview/resource/R1J12aFuGROf1

Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Film nawiązujący do treści materiału. Wprowadzenie do tablic - tablice jednowymiarowe. Realizacja tablic w języku Python.

Kod programu zaprezentowanego w filmie:

Plik o rozmiarze 1.07 KB w języku polskim

Słownik

f-string

mechanizm formatowania łańcuchów znaków, dokładnie opisany w dokumencie PEP 498 ; wprowadzony w języku Python w wersji `3.6`, znany również jako *Literal String Interpolation*; zakłada format: `f"Napis, a w nim {zmienna} do wypisania"`.

krotka, tupla

(w języku Python `tuple`) sekwencyjny typ danych, który przechowuje elementy różnego typu, np. `(1, "A", 3.14)`; od listy różni się tym, że nie jest modyfikowalny

lista

(w języku Python `list`) sekwencyjny typ danych, który przechowuje elementy różnego typu, np. `[1,"A",3.14]`; może być modyfikowany

niezmienna

(ang. *immutable*) sekwencja, która jest niemodyfikowalna „w miejscu”; nie można zmienić żadnego z jej elementów wewnątrz – trzeba stworzyć nowy obiekt, który zawiera zmienione elementy

obiekt iterowany

(ang. *iterable*) obiekt, który ma budowę sekwencyjną; może być przekazywany do pętli `for` w celu wyodrębnienia z niego pojedynczych elementów; obiektami iterowanymi są `list, str, tuple` – przykładowo, `[1,3,5]`, `'Python'` lub `(3, 6, 'A')`

wyjątek

(ang. *exception*) sposób sygnalizowania przez program sytuacji wyjątkowych (najczęściej błędów); w momencie wyrzucenia wyjątku musi on zostać obsłużony, w przeciwnym wypadku program kończy pracę

wyrażenie indeksujące

(ang. *slice*) nazywane także `wycinkiem`, zawarte w nawiasach kwadratowych wyrażenie, które pozwala określić wybraną część sekwencji; może mieć ono postać `[a:b]`, `[a]` albo `[a:b:c]`

wyrażenie listowe

(ang. *list compehension*) wyrażenie, dzięki któremu możemy wygenerować listę zawierającą określone elementy bez konieczności użycia pętli; przykład wyrażenia

listowego generującego listę zawierającą kwadraty liczb od 0 do 10:

```
[x**2 for x in range(11)]
```

zmienna (modyfikowalna, zmienialna)

(ang. *mutable*) zestaw danych, którego elementy da się zmienić bezpośrednio, bez konieczności tworzenia nowego obiektu i przypisywania wartości jego elementom

Prezentacja multimedialna

Rozwiążemy teraz pewien problem, używając tablic.

Polecenie 1

Napisz program, który dla danych dwóch niepustych, posortowanych nierosnąco tablic, wypełnionych liczbami całkowitymi, utworzy tablicę zawierającą elementy tablic bez powtórzeń w kolejności rosnącej.

Działanie programu przetestuj dla przykładowych tablic:

```
1 tablica_1 = [30, 25, 22, 21, 19, 18, 14, 14, 13, 9, 8, 7, 7, 5,
2 tablica_2 = [29, 25, 21, 17, 14, 12, 10, 10, 10, 8, 6, 5, 4, 4,
```

Specyfikacja problemu:

Dane:

- `tablica_1, tablica_2` – niepuste, posortowane nierosnąco tablice, zawierające wyłącznie liczby całkowite

Wynik:

- `wynikowa_lista` – tablica zawierająca elementy z wejściowych tablic, bez powtórzeń, w kolejności rosnącej

Wyniki dla tablic z przykładu:

```
1 [-10,-7,-5,-1,1,2,3,4,5,6,7,8,9,10,12,13,14,17,18,19,21,22,25,2
```

Twoje zadania

1. Napisz program, który dla danych dwóch niepustych, posortowanych nierosnąco tablic, wypełnionych liczbami całkowitymi, utworzy tablicę zawierającą elementy tablic bez powtórzeń w kolejności rosnącej.

```
1 def unikalne_niemalejace_wartosci(tablica_1, tablica_2):
2     wynikowa_lista = []
3
```

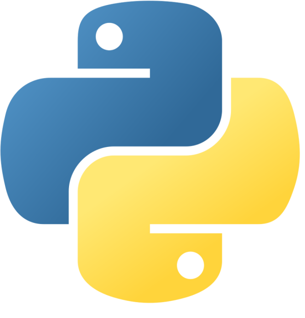
```
4       # tutaj dodaj swój kod
5
6       return wynikowa_lista
7
8   tablica_1 = [30, 25, 22, 21, 19, 18, 14, 14, 13, 9, 8, 7, 7,
    5, 3, 2, -1, -10]
9   tablica_2 = [29, 25, 21, 17, 14, 12, 10, 10, 10, 8, 6, 5, 4,
    4, 3, 1, -5, -7]
10
11  print(unikalne_niemalejace_wartosci(tablica_1, tablica_2))
```

```
1
```

Polecenie 2

Porównaj swoje rozwiązanie z prezentacją.

1



Pamiętaj, że struktury zwane tablicami w języku Python implementuje się za pomocą list.
Źródło: Wikimedia Commons, wikimedia.commons.org, dostęp 14.11.2022, licencja tylko do użytku edukacyjnego.

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PTkqJknAk

Przeprowadźmy analizę problemu. Zadaniem programu jest scalenie dwóch uporządkowanych nierosnąco tablic. Ważne, aby w wynikowej tablicy nie powtarzały się elementy i aby była ona uporządkowana rosnąco.

Definiujemy funkcję o nazwie `unikalne_niemalejace_wartosci` z dwoma parametrami: `tablica_1` oraz `tablica_2`. Instrukcję `pass` wykorzystamy w miejscach, gdzie jeszcze nie zdążyliśmy zaimplementować potrzebnego fragmentu, aby interpreter nie zwracał błędu.

```
1 def unikalne_niemalejace_wartosci(tablica_1, tablica_2):
2     pass
3
```

2

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PTkqJknAk

W kolejnym kroku deklarujemy zmienne `indeks_1` oraz `indeks_2`, za pomocą których iterujemy po tablicach oraz tablicę, którą funkcja będzie zwracała – `wynikowa_lista`. Początkową wartość indeksów ustawiamy na ostatni indeks każdej z tablic. Dzięki temu, jeżeli będziemy przechodzić po nich od końca do początku, przejdziemy po ich elementach w kolejności niemalejącej (wejściowe tablice są posortowane nierosnąco). Ponieważ w funkcji pojawił się kod, możemy usunąć `pass`.

```
1 def
  unikalne_niemalejace_warto
  sci(tablica_1,tablica_2):
2     indeks_1 =
  len(tablica_1) - 1
3     indeks_2 =
  len(tablica_2) - 1
4     wynikowa_lista = []
5
```

3

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PTkqJknAk

Kolejnym krokiem jest stworzenie pętli, która będzie iterować po elementach tablicy. Wykorzystamy do tego pętlę `while`

. Warunek zakończenia dla pętli stanowi zmiana wartości któregoś z indeksów na -1. Oznacza to, że przeszliśmy po wszystkich indeksach jednej z tablic.

```
1 def
  unikalne_niemalejace_warto
  sci(tablica_1, tablica_2):
2     indeks_1 =
  len(tablica_1) - 1
```

```
3     indeks_2 =
len(tablica_2) - 1
4     wynikowa_lista = []
5
6     while indeks_1 != -1
and indeks_2 != -1:
7         pass
8
```

4

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PTkqJknAk

W pętli `while`

chcemy przetworzyć wejściowe tablice i stworzyć z nich tablicę wyjściową. Przypomnijmy, że iterujemy po elementach każdej tablicy w kolejności od jej końca do początku. Pętla zakończy się, kiedy przejdziemy po wszystkich elementach jednej z tablic. Zauważmy, że nie ma możliwości, aby przechodzenie zakończyło się dla obu tablic jednocześnie, ponieważ zawsze pobieramy element tylko z jednej tablicy. Przy każdej iteracji pętli bierzemy najmniejszy ze wskazywanych przez obecne indeksy element. Oczywiście, jeżeli wykorzystamy element z którejś tablicy, musimy zmniejszyć indeks dla tej tablicy, aby wskazywał następny element do przetworzenia.

```
1 def
unikalne_niemalejace_warto
sci(tablica_1, tablica_2):
2     indeks_1 =
len(tablica_1) - 1
3     indeks_2 =
len(tablica_2) - 1
4     wynikowa_lista = []
5
```

```
    while indeks_1 != -1 and indeks_2 != -1:
        if tablica_1[indeks_1] <= tablica_2[indeks_2]:
            minimalny_element = tablica_1[indeks_1]
            indeks_1 -= 1
        else:
            minimalny_element = tablica_2[indeks_2]
            indeks_2 -= 1


```

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PTkqJknAk

Znaleziony minimalny element dodajemy do wynikowej tablicy. Jest jeden warunek – przed dodaniem sprawdzamy, czy nie jest to ten sam element, co ostatni dodany do tablicy. W ten sposób mamy pewność, że wynikowa tablica zawiera jedynie unikalne elementy. Jeżeli pobieramy zawsze najmniejszy ze wskazywanych przez indeksy elementów, a tablice są posortowane nierosnąco, to każdy element przy pierwszym pojawieniu się zostaje dodany, natomiast całą resztę odrzucamy.

Dodajemy również sprawdzenie, czy `wynikowa_lista` nie jest pusta. Taki przypadek występuje przy pierwszej iteracji pętli `while`. Gdybyśmy go nie dodali, sprawdzając ostatni dodany do wynikowej tablicy element, użylibyśmy indeksu, który jest poza zakresem tablicy i program zasygnalizowałby `IndexError`. Dodatkowy warunek umieścimy

w wyrażeniu logicznym `or`, razem z warunkiem sprawdzenia ostatniego elementu. Ważne jest, aby warunek sprawdzający, czy tablica jest pusta, był w wyrażeniu po lewej stronie, ponieważ w języku Python ewaluacja wyrażeń logicznych przebiega od lewej do prawej strony.

```python
def unikalne_niemalejace_wartosci(tablica_1, tablica_2):
    indeks_1 = len(tablica_1) - 1
    indeks_2 = len(tablica_2) - 1
    wynikowa_lista = []

    while indeks_1 != -1 and indeks_2 != -1:
        if tablica_1[indeks_1] <= tablica_2[indeks_2]:
            minimalny_element = tablica_1[indeks_1]
            indeks_1 -= 1
        else:
            minimalny_element = tablica_2[indeks_2]
            indeks_2 -= 1

        if len(wynikowa_lista) == 0 or minimalny_element != wynikowa_lista[-1]:
            wynikowa_lista.append(minimalny_element)


```

6

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PTkqJknAk

Pętla `while` jest już gotowa. Kiedy program ją opuszcza, elementy jednej z wejściowych tablic są całkowicie wykorzystane. Ustalamy, która z tablic została wykorzystana, a następnie dodajemy elementy z drugiej tablicy do wynikowej listy. Pamiętamy jednak, że w pozostałej tablicy mogą znajdować się powtarzające się elementy, więc ponownie porównamy je z ostatnim elementem listy.

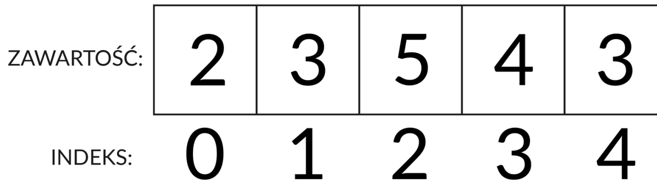
Aby wskazać tablicę, której elementy się skończyły, sprawdzamy, który indeks jest równy `-1`. Zależnie od tego uruchamiamy pętlę `while` dla pierwszej lub drugiej tablicy. Warunkiem zakończenia każdej z pętli jest zmiana indeksu dla iterowanej tablicy na `-1`.

```
def unikalne_niemalejace_wartosci(tablica_1, tablica_2):
    indeks_1 = len(tablica_1) - 1
    indeks_2 = len(tablica_2) - 1
    wynikowa_lista = []

    while indeks_1 != -1 and indeks_2 != -1:
        if tablica_1[indeks_1] <= tablica_2[indeks_2]:
            minimalny_element = tablica_1[indeks_1]
            indeks_1 -= 1
        else:
            minimalny_element = tablica_2[indeks_2]
            indeks_2 -= 1

```

```
14        if len(wynikowa_lista) == 0 or minimalny_element != wynikowa_lista[-1]:
15            wynikowa_lista.append(minimalny_element)
16
17    if indeks_1 == -1:
18        while indeks_2 != -1:
19            if tablica_2[indeks_2] != wynikowa_lista[-1]:
20                wynikowa_lista.append(tablica_2[indeks_2])
21            indeks_2 -= 1
22    else:
23        while indeks_1 != -1:
24            if tablica_1[indeks_1] != wynikowa_lista[-1]:
25                wynikowa_lista.append(tablica_1[indeks_1])
26            indeks_1 -= 1
27
28
```



Elementy tablicy indeksuje się od zera, zatem ostatni element tablicy zawsze ma indeks o jeden mniejszy niż długość tablicy.
Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PTkqJknAk

Za pomocą nowych pętli `while` przetworzyliśmy wszystkie elementy tablic, zwracamy więc wynikową listę.

```
1  def unikalne_niemalejace_wartosci(tablica_1, tablica_2):
2      indeks_1 = len(tablica_1) - 1
3      indeks_2 = len(tablica_2) - 1
4      wynikowa_lista = []
5
6      while indeks_1 != -1 and indeks_2 != -1:
7          if tablica_1[indeks_1] <= tablica_2[indeks_2]:
8              minimalny_element = tablica_1[indeks_1]
9              indeks_1 -= 1
10         else:
11             minimalny_element = tablica_2[indeks_2]
12             indeks_2 -= 1
13
14         if len(wynikowa_lista) == 0 or minimalny_element != wynikowa_lista[-1]:
15             wynikowa_lista.append(minimalny_element)
16
17     if indeks_1 == -1:
18         while indeks_2 != -1:
19             if tablica_2[indeks_2] != wynikowa_lista[-1]:
```

```
20
   wynikowa_lista.append(tabl
   ica_2[indeks_2])
21             indeks_2 -= 1
22     else:
23         while indeks_1 !=
   -1:
24             if
   tablica_1[indeks_1] !=
   wynikowa_lista[-1]:
25
   wynikowa_lista.append(tabl
   ica_1[indeks_1])
26             indeks_1 -= 1
27
28     return wynikowa_lista
29
30 tablica_1 = [30, 25, 22,
   21, 19, 18, 14, 14, 13, 9,
   8, 7, 7, 5, 3, 2, -1, -10]
31 tablica_2 = [29, 25, 21,
   17, 14, 12, 10, 10, 10, 8,
   6, 5, 4, 4, 3, 1, -5, -7]
32
33 print(unikalne_niemalejace
   _wartosci(tablica_1,
   tablica_2))
```

Źródło: Contentplus.pl Sp. z o.o., licencja: CC BY-SA 3.0.

Polecenie 3

Zmodyfikuj omówioną w prezentacji implementację tak, aby po wyjściu z pierwszej pętli `while`, nie używać dwóch instrukcji warunkowych z pętlą, lecz jednej instrukcji warunkowej i jednej pętli.

Sprawdź się

Pokaż ćwiczenia:

Ćwiczenie 1

Ćwiczenie 2

Ćwiczenie 3

Napisz program, który z podanej listy liczb całkowitych wybierze tylko te, które są potęgami liczby 3 i utworzy z nich nową listę (pamiętaj, że 1 również jest potęgą liczby 3). Załóż, że w wejściowej liście występują liczby mniejsze od 100000.

Przetestuj działanie programu dla listy:

```
1 lista_danych = [789, 499, 746, -50180, 2, 38, 98774, -3, 4600
```

Specyfikacja problemu:

Dane:

- `lista_danych` – lista zawierająca liczby całkowite mniejsze od 100000

Wynik:

- `wynikowa_lista` – lista zawierająca te elementy `lista_danych`, które są potęgami liczby 3

Dla nauczyciela

Autor: Adam Jurkiewicz
Przedmiot: Informatyka

Temat: Tablice jednowymiarowe w języku Python

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

I. Rozumienie, analizowanie i rozwiązywanie problemów na bazie logicznego i abstrakcyjnego myślenia, myślenia algorytmicznego i sposobów reprezentowania informacji.

II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

Treści nauczania – wymagania szczegółowe

I. Rozumienie, analizowanie i rozwiązywanie problemów.

Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

2) do realizacji rozwiązania problemu dobiera odpowiednią metodę lub technikę algorytmiczną i struktury danych;

I + II. Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:

2) wykorzystuje znane sobie algorytmy przy rozwiązywaniu i programowaniu rozwiązań następujących problemów:

c) znajdowania w ciągu podciągów o różnorodnych własnościach, np. najdłuższego spójnego podciągu niemalejącego, spójnego podciągu o największej sumie,

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;

- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Określisz, w jaki sposób w języku Python implementuje się tablice.
- Prześledzisz, jak definiuje się listy oraz krotki.
- Przeanalizujesz algorytm przeszukiwania listy i szukania najdłuższego spójnego niemalejącego podciągu.
- Skonstruujesz algorytm i zaimplementujesz program przeszukujący zestaw danych.
- Scharakteryzujesz metody specyficzne dla struktury `list` w języku Python.
- Wymienisz zastosowania list oraz krotek.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- oprogramowanie dla języka Python 3 (lub nowszej wersji), w tym PyCharm lub IDLE.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Tablice jednowymiarowe w języku Python”. Nauczyciel prosi uczniów o zapoznanie się z multimedium w sekcji „Prezentacja multimedialna”.

Faza wstępna:

1. Nauczyciel wyświetla temat i cele zajęć zawarte w sekcji „Wprowadzenie”. Następnie wspólnie z uczniami ustala kryteria sukcesu.
2. Problem – uczniowie szukają spójnego podciągu rosnącego w tablicy liczb. Przykładowa tablica: [12, 8, 7, 7, 8, 7, 6, 6, 5, 3, 4, 5, 5, 5, 5, 1, 2, 3, 4, 3]

Faza realizacyjna:

1. **Praca z tekstem.** Nauczyciel ocenia, na podstawie informacji na platformie, stan przygotowania uczniów do zajęć. Jeżeli jest ono niewystarczające, prosi wybraną osobę o przedstawienie najważniejszych informacji z sekcji „Przeczytaj”. Następnie uczniowie rozwiązują problem 1 z tej sekcji. Chętne lub wybrane osoby prezentują swoje rozwiązanie na forum klasy, nauczyciel je omawia.
2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Prezentacja multimedialna”. Uczniowie pracując w parach, rozwiązują polecenie 1. W kolejnym kroku uczniowie porównują swoje programy z tymi omówionymi w prezentacji. W razie potrzeby nauczyciel wyjaśnia niezrozumiałe kwestie i wątpliwości.
3. **Ćwiczenie umiejętności.** Uczniowie wykonują ćwiczenia nr 1 i 2 z sekcji „Sprawdź się”. Nauczyciel sprawdza poprawność wykonanych zadań, omawiając je wraz z uczniami.
4. Uczniowie dobierają się w pary i wykonują ćwiczenie nr 3 z sekcji „Sprawdź się”. Następnie konsultują swoje rozwiązanie z inną parą uczniów.

Faza podsumowująca:

1. Nauczyciel wyświetla na tablicy temat lekcji i cele zawarte w sekcji „Wprowadzenie”. W kontekście ich realizacji podsumowuje przebieg zajęć, a także wskazuje mocne i słabe strony pracy uczniów.
2. Wybrany uczeń podsumowuje zajęcia, zwracając uwagę na nabyte umiejętności.

Praca domowa:

1. Uczniowie wykonują polecenie 3 z sekcji „Prezentacja multimedialna”.

Wskazówki metodyczne:

- Przed lekcją uczniom można zaproponować powtórzenie wiadomości na temat struktur, które zostały zaprezentowane w początkowych e-materiałach.