Podstawowe elementy języka

Prymitywne typy danych, kolekcje, instrukcje sterujące

Adam Chyła
chyla.org

Podstawowe typy danych

Typy informują interpreter o wewnętrznym sposobie reprezentacji danych oraz operacjach, które można na nich wykonywać. Spośród wszystkich dostępnych typów wbudowanych warto wyróżnić cztery podstawowe:

- str,
- bool,
- int,
- float.

Podstawowe typy danych

Napisy (str - string)
są szeroko wykorzystywane w programowaniu, większość skryptów wykorzystuje je do zapisu danych czy komunikacji z innymi systemami.

Napisy typu `str` (string) tworzone są najczęściej poprzez ujęcie ciągu znaków w apostrof (') lub podwójny apostrof (").

```
napis = 'Ala ma kota'
napis2 = "Ala ma kota"

print(napis)
print(napis2)
```

Podstawowe typy danych

Umieszczenie tekstu pomiędzy trzykrotnie powtórzonym znakiem apostrofu lub podwójnego apostrofu pozwala na stworzenie napisu złożonego z wielu linii.

```
napis_wiloliniowy = '''Ala
ma
kota'''
napis_wiloliniowy2 = """Ala
ma
kota"""

print(napis_wiloliniowy)
print(napis_wiloliniowy2)
```

Podstawowe typy danych

Typ logiczny (bool – boolean) określany jest przez wartości `True` oraz `False`, wykorzystuje się go do weryfikacji warunków i przeprowadzania operacji logicznych.

Poszczególne wyrażenia mogą być ze sobą łączone za pomocą **alternatywy** (`or`), **koniunkcji** (`and`) oraz **negacji** (`not`).

```
wynik = (1 + 1 == 2)
print(wynik)

wynik = (1 + 1 == 2 and not 0 > 1)
print(wynik)
```

Podstawowe typy danych

Typ liczb całkowitych (int – integer) określa liczby o dowolnej wielkości.
W wyniku przeprowadzanych operacji matematycznych możliwe jest otrzymanie liczby **typu rzeczywistego (float)**.

```
a = 10
b = 2

# w wyniku mnożenia otrzymujemy liczbę typu int
print("a * b =", a * b)

# w wyniku dzielenia otrzymujemy liczbę typu float
print("a / b =", a / b)
```

Podstawowe typy danych

Do **konwersji danych danego typu na inny** wykorzystywana jest nazwa typu docelowego taka jak `str`, `bool`, `int`, `float`.

```
# str(x) --> zamiana x na napis
napis = str(1)  # napis = "1"

# int(x) --> zamiana x na liczbę
liczba = int("2")  # liczba = 2

# float(x) --> zamiana x na liczbę typu float
rzeczywista = float("2.3")  # rzeczywista = 2.3

# bool(x) --> zamiana x na wartość logiczną
logiczna = bool(0)      # logiczna = False
logiczna = bool(1)      # logiczna = True
logiczna = bool("")     # logiczna = False
logiczna = bool("abc")  # logiczna = True
```

Podstawowe typy danych

Wbudowana **funkcja `type()`** może posłużyć jako narzędzie do określania typu podanego wyrażenia.

```
a = 20
b = 2.0
c = a * b
print("Typ a", type(a))
print("Typ b:", type(b))
print("Typ c:", type(c))

Wynik działania powyższego skryptu:
Typ a: <class 'int'>
Typ b: <class 'float'>
Typ c: <class 'float'>
```

Kolekcje

Kolekcje służą do przechowywania zestawów danych, spośród wszystkich dostępnych typów na uwagę zasługują cztery najczęściej używane:

- list,
- tuple,
- set,
- dict.

Kolekcje

Lista (`list`) służy do przechowywania zestawu danych. Kolejność elementów listy jest zależna od kolejności dodawania elementów.

Elementy przechowywane w liście nie muszą być tego samego typu (np. ta sama lista może przechowywać napisy jak i liczby).

```
# utworzenie pustej listy
moja_lista = []
moja_lista = list()

# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
#    element  -7 -6  -5   -4   -3  -2 -1
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]

# wyświetlenie na ekranie drugiego elementu
print(moja_lista[1])

# wyświetlenie na ekranie przedostatniego elementu
print(moja_lista[5])
print(moja_lista[-2])
```

Kolekcje

Częste operacje na listach:

- sprawdzenie długości:
  wolna funkcja `len()`
- dodanie elementu:
  funkcja `append()`
- wstawienie elementu:
  funkcja `insert()`
- usunięcie elementu:
  funkcja `remove()`
  wolna funkcja `del()`
- wydzielenie kolekcji:
  slicing

```
# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]
```

Kolekcje

Częste operacje na listach:

- sprawdzenie długości: **wolna funkcja `len()`**
- dodanie elementu: funkcja `append()`
- wstawienie elementu: funkcja `insert()`
- usunięcie elementu: funkcja `remove()` wolna funkcja `del()`
- wydzielenie kolekcji: slicing

```
# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]

# otrzymanie liczby elementów listy
rozmiar_listy = len(moja_lista)
print("rozmiar_listy =", rozmiar_listy)
```

Kolekcje

Częste operacje na listach:

- sprawdzenie długości: wolna funkcja `len()`
- dodanie elementu: **funkcja `append()`**
- wstawienie elementu: funkcja `insert()`
- usunięcie elementu: funkcja `remove()` wolna funkcja `del()`
- wydzielenie kolekcji: slicing

```
# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]

# dodanie elementu do listy
moja_lista.append(5)
print("moja_lista = ", moja_lista)

Wynik działania powyższego skryptu:
moja_lista = [1, 2, 'a', 'b', 'c', 3, 4, 5]
```

Kolekcje

Częste operacje na listach:

- sprawdzenie długości: wolna funkcja `len()`
- dodanie elementu: funkcja `append()`
- wstawienie elementu: **funkcja `insert()`**
- usunięcie elementu: funkcja `remove()` wolna funkcja `del()`
- wydzielenie kolekcji: slicing

```
# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]

# wstawienie elementu w określone miejsce listy
moja_lista.insert(2, 'nowa')
print("moja_lista = ", moja_lista)

Wynik działania powyższego skryptu:
moja_lista =  [1, 2, 'nowa', 'a', 'b', 'c', 3, 4]
```

Kolekcje

Częste operacje na listach:

- sprawdzenie długości: wolna funkcja `len()`
- dodanie elementu: funkcja `append()`
- wstawienie elementu: funkcja `insert()`
- usunięcie elementu: **funkcja `remove()`** wolna funkcja `del()`
- wydzielenie kolekcji: slicing

```
# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]

# usunięcie pierwszego elementu listy,
# którego wartość jest równa wartości elementu
# przekazanego do funkcji remove
moja_lista.remove('b')
print("moja_lista = ", moja_lista)

Wynik działania powyższego skryptu:
moja_lista =  [1, 2, 'a', 'c', 3, 4]
```

Kolekcje

Częste operacje na listach:

- sprawdzenie długości: wolna funkcja `len()`
- dodanie elementu: funkcja `append()`
- wstawienie elementu: funkcja `insert()`
- usunięcie elementu: funkcja `remove()` **wolna funkcja `del()`**
- wydzielenie kolekcji: slicing

```
# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]

# usunięcie określonego elementu listy
del(moja_lista[2])
print("moja_lista = ", moja_lista)

Wynik działania powyższego skryptu:
moja_lista =  [1, 2, 'b', 'c', 3, 4]
```

Kolekcje

Częste operacje na listach:

- sprawdzenie długości: wolna funkcja `len()`
- dodanie elementu: funkcja `append()`
- wstawienie elementu: funkcja `insert()`
- usunięcie elementu: funkcja `remove()` wolna funkcja `del()`
- **wydzielenie kolekcji: slicing**

```
# utworzenie listy zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_lista = [ 1, 2, 'a', 'b', 'c', 3, 4 ]

# wydzielenie nowej listy na podstawie istniejącej
nowa_lista = moja_lista[1:5]
print("moja_lista = ", moja_lista)
print("nowa_lista = ", nowa_lista)

Wynik działania powyższego skryptu:
moja_lista =  [1, 2, ‘a’, 'b', 'c', 3, 4]
nowa_lista =  [2, 'a', 'b', 'c']
```

Kolekcje

Krotka (`tuple`), podobnie jak lista, służy do przechowywania zestawu danych. Kolejność elementów jest zależna od kolejności zdefiniowanej w chwili utworzenia krotki.

Elementy przechowywane w krotce nie muszą być tego samego typu (np. ta sama krotka może przechowywać napisy jak i liczby).

```
# utworzenie pustej krotki
moja_krotka = ()
moja_krotka = tuple()

# utworzenie krotki zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
#    element  -7 -6  -5   -4   -3  -2 -1
moja_krotka = ( 1, 2, 'a', 'b', 'c', 3, 4 )

# wyświetlenie na ekranie drugiego elementu
print(moja_krotka[1])

# wyświetlenie na ekranie przedostatniego elementu
print(moja_krotka[5])
print(moja_krotka[-2])
```

Kolekcje

Raz utworzona krotka nie może być później zmieniona.

```
# utworzenie krotki zawierającej 7 elementów
#    element   0  1   2    3    4   5  6
moja_krotka = ( 1, 2, 'a', 'b', 'c', 3, 4 )

# próba usunięcia elementu z krotki
del(moja_krotka[1]) # BŁĄD!
```

Kolekcje

Napisy (`str`) zachowują się podobnie do krotek:

- są niemodyfikowalne,
- do poszczególnych liter można odwołać się za pomocą indeksu,
- funkcja `len()` zwraca liczbę liter,
- wspierają tworzenie napisów częściowych (slicing).

```
moj_napis = "Ala ma kota"

# wyświetlenie na ekranie trzeciego elementu
print(moj_napis[2])

# wyświetlenie na ekranie przedostatniego elementu
print(moj_napis[-2])

# utworzenie nowego elementu, slicing
kot = moj_napis[7:]    # brak liczby po dwukropku!
print("kot =", kot)
```

Kolekcje

Listy, krotki i napisy pozwalają na wykonanie operacji dających podobne rezultaty na każdym z tych typów. Do operacji tych możemy zaliczyć między innymi omówione wcześniej:

- odwoływanie się do poszczególnych elementów za pomocą indeksów (operator indeksowania, `[]`),
- pobieranie liczby elementów przechowywanych w danym typie (funkcja `len()`),
- wydzielenie nowej kolekcji na podstawie kolekcji istniejącej (slicing, `[x:y]`).

Ważny wniosek:

Listy, krotki i napisy posiadają pewien wspólny interfejs (zbiór pewnych funkcji), który pozwala tym typom zachowywać się w podobny sposób.

Kolekcje

Zbior (set) służy do przechowywania unikalnego zestawu danych bez zachowania kolejności elementów.

W związku z niezachowywaniem kolejności elementów w zbiorze **nie jest możliwe odwoływanie się do poszczególnych elementów za pomocą operatora indeksowania**.

```
# utworzenie pustego zbioru
zbior_A = set()

# utworzenie zbioru z 4 elementami
zbior_B = { 1, 1, 99, 2, 3 }

print("zbior_B =", zbior_B)

print("zbior_B[1] = ", zbior_B[1]) # BŁĄD!
```

Kolekcje

Najczęściej wykorzystywane funkcje do operacji na zbiorach:

- dodanie nowego elementu: funkcja `add()`
- usunięcie elementu: funkcja `remove()`
- suma zbiorów: operator `|`
- część wspólna zbiorów: operator `&`
- różnica zbiorów: operator `-`

```
zbior_A = set()
zbior_B = { 1, 1, 99, 2, 3 }


print("zbior_A =", zbior_A)
print("zbior_B =", zbior_B)
```

Kolekcje

Najczęściej wykorzystywane funkcje do operacji na zbiorach:

- **dodanie nowego elementu: funkcja `add()`**
- usunięcie elementu: funkcja `remove()`
- suma zbiorów: operator `|`
- część wspólna zbiorów: operator `&`
- różnica zbiorów: operator `-`

```
zbior_A = set()
zbior_B = { 1, 1, 99, 2, 3 }

# dodanie elementów
zbior_A.add(1)
zbior_A.add(1)
zbior_A.add(2)

print("zbior_A =", zbior_A)
print("zbior_B =", zbior_B)
```

Kolekcje

Najczęściej wykorzystywane funkcje do operacji na zbiorach:

- dodanie nowego elementu: funkcja add()
- **usunięcie elementu: funkcja remove()**
- suma zbiorów: operator |
- część wspólna zbiorów: operator &
- różnica zbiorów: operator -

```
zbior_A = set()
zbior_B = { 1, 1, 99, 2, 3 }

# usunięcie elementu
zbior_B.remove(99)

print("zbior_A =", zbior_A)
print("zbior_B =", zbior_B)
```

Kolekcje

Najczęściej wykorzystywane funkcje do operacji na zbiorach:

- dodanie nowego elementu: funkcja `add()`
- usunięcie elementu: funkcja `remove()`
- **suma zbiorów: operator |**
- część wspólna zbiorów: operator &
- różnica zbiorów: operator -

```
zbior_A = { 99, 3, 5, 9 }
zbior_B = { 1, 1, 99, 2, 3 }

# suma zbiorów
zbior_C = zbior_A | zbior_B

print("zbior_A =", zbior_A)
print("zbior_B =", zbior_B)
print("zbior_C =", zbior_C)
```

Kolekcje

Najczęściej wykorzystywane funkcje do operacji na zbiorach:

- dodanie nowego elementu: funkcja `add()`
- usunięcie elementu: funkcja `remove()`
- suma zbiorów: operator `|`
- **część wspólna zbiorów: operator &**
- różnica zbiorów: operator `-`

```
zbior_A = { 99, 3, 5, 9 }
zbior_B = { 1, 1, 99, 2, 3 }

# część wspólna
zbior_C = zbior_A & zbior_B

print("zbior_A =", zbior_A)
print("zbior_B =", zbior_B)
print("zbior_C =", zbior_C)
```

Kolekcje

Najczęściej wykorzystywane funkcje do operacji na zbiorach:

- dodanie nowego elementu: funkcja `add()`
- usunięcie elementu: funkcja `remove()`
- suma zbiorów: operator `|`
- część wspólna zbiorów: operator `&`
- **różnica zbiorów: operator `-`**

```
zbior_A = { 99, 3, 5, 9 }
zbior_B = { 1, 1, 99, 2, 3 }

# różnica zbiorów - wszystkie elementy
# ze zbioru A bez elementów ze zbioru B
zbior_C = zbior_A - zbior_B

print("zbior_A =", zbior_A)
print("zbior_B =", zbior_B)
print("zbior_C =", zbior_C)
```

Kolekcje

Słownik (dict) służy do przechowywania zestawów par klucz - wartość. Kluczem mogą być wartości, których nie można zmieniać (np. napisy, liczby, krotki).

Utworzenie nowego elementu oraz odwołanie się do istniejącego elementu słownika odbywa się za pomocą operatora indeksowania.

```
# utworzenie pustego słownika
kolory = {}

# utworzenie słownika z 3 elementami
kolory = {
    "#000000" : "czarny",
    "#FFFFFF" : "bialy",
    "#808080" : "szary",
}

# dodanie elementu
kolory["#FF3333"] = "czerwony"

# odwołanie do elementu
nazwa_koloru = kolory["#FFFFFF"]
```

Kolekcje

Podobnie jak było to w przypadku typów podstawowych, do **konwersji danych jednego typu na inny** wykorzystywana jest nazwa typu docelowego taka jak `list, tuple, set`.

```
# tuple(x) --> zamiana x na krotkę
krotka = tuple("aabbcc")

# list(x) --> zamiana x na listę
lista = list(krotka)

# set(x) --> zamiana x na zbiór
zbior = set(lista)

# zamiana kolekcji na napis następuje
# przy użyciu funkcji join()
unikalne = tuple(zbior)
napis = ''.join(unikalne) # kolejność liter dowolna
```

Kolekcje

Uniwersalny operator **in** pozwala sprawdzić czy dany element znajduje się w kolekcji.

```
krotka = (1, 2, 4, 5, 6)

czy_dwa = 2 in krotka
czy_trzy = 3 in krotka

print("czy_dwa =", czy_dwa)
print("czy_trzy =", czy_trzy)

Wynik działania powyższego skryptu:
czy_dwa = True
czy_trzy = False
```

Instrukcje sterujące

Programy nie zawsze wykonują te same czynności, czasami ich dalsze zachowanie jest uzależnione od pewnych warunków. Przykładowo, aplikacja kontrolująca działanie robota sprzątającego musi zmienić kierunek ruchu odkurzacza, gdy wykryje, że urządzenie uderzyło w przeszkodę. Do sterowania przebiegiem wykonania programu wykorzystywana jest instrukcja warunkowa (if), pętla for oraz while.

Instrukcje sterujące

Instrukcja warunkowa pozwala wykonać fragment kodu, gdy zostanie spełniony określony warunek.

Istotne elementy:

- słowo kluczowe `if`,
- dwukropek na końcu linii,
- widoczne wcięcie (**4 spacje**) oznaczające blok kodu do wykonania.

```
imie = input("Podaj swoje imie: ")

if imie == "Jan":
    print("Twoje imie to Jan")
elif imie == "Mateusz":
    print("O Mateusz!")
else:
    print("Nie znamy sie jeszcze")

print("Dzien dobry!")
```

Instrukcje sterujące

Uwaga!

Dwukropek na końcu linii oznacza, że w kolejnej linii następuje odpowiednio wcięty blok kodu złożony z co najmniej jednej linii.

Brak wcięcia oznacza brak bloku kodu do wykonania, co spowoduje przerwanie pracy programu i zgłoszenie błędu przez interpreter.

```
imie = input("Podaj swoje imie: ")

if imie == "Jan":
    print("Twoje imie to Jan")
elif imie == "Mateusz":
    print("O Mateusz!")
else:
    print("Nie znamy sie jeszcze")

print("Dzien dobry!")
```

Instrukcje sterujące

W języku Python dostępna jest specjalna **wartość None**, gdy zmienna przyjmuje taką wartość oznacza to, że nie przechowuje ona żadnych danych.

Sprawdzenia tej wartości dokonujemy za pomocą operatora `is`.

```
a = None
if a is None:
    print("Zmienna a nie ma ustawionej wartosci")
else:
    print("a:", a)
```

Instrukcje sterujące

Konstrukcja while pozwala na ponowne wykonywanie danego fragmentu kodu, dopóki podany warunek jest spełniony. Wykonywanie pętli zostanie zatrzymane, gdy warunek przestanie być prawdziwy.

Istotne elementy:

- słowo kluczowe `while`,
- dwukropek na końcu linii,
- widoczne wcięcie (**4 spacje**) oznaczające blok kodu do wykonania.

```
print("Zaczynam liczyc")

i = 1
while i < 6:
    print("Liczba:", i)
    i = i + 1

print("Koncze liczyc")
```

Instrukcje sterujące

Powtarzanie tych samych fragmentów kodu możliwe jest także za pomocą **pętli for**. Pętla ta pozwala na wybranie kolejnych elementów z podanej kolekcji.

Istotne elementy:

- słowo kluczowe `if` oraz `in`,
- dwukropek na końcu linii,
- widoczne wcięcie (**4 spacje**) oznaczające blok kodu do wykonania.

```
print("Zaczynam")

for nazwa in ["Ala", "ma", 2, "koty"]:
    print("Element:", nazwa)

print("Koncze")

Wynik działania powyższego skryptu:
Zaczynam
Element: Ala
Element: ma
Element: 2
Element: koty
Koncze
```

Instrukcje sterujące

Pętla for służy do iterowania (przechodzenia) po kolejnych elementach pewnego zestawu danych. W skład tego zestawu (np. listy lub krotki) mogą wchodzić kolejne liczby naturalne.

```
print("Zaczynam")

for nazwa in (1, 2, 3, 4):
    print("Element:", nazwa)

print("Koncze")

Wynik działania powyższego skryptu:
Zaczynam
Element: 1
Element: 2
Element: 3
Element: 4
Koncze
```

Instrukcje sterujące

Korzystanie z kolejnych liczb w pętli for jest dość częste. W przypadku próby iteracji po liczbach od 0 do 100 wykorzystanie listy byłoby uciążliwe, z tego powodu powstała **funkcja range**, która tworzy obiekt zwracający kolejne liczby całkowite.

```
print("Zaczynam")

for nazwa in range(1, 5):      # przedział <1, 5)
    print("Element:", nazwa)

print("Koncze")

Wynik działania powyższego skryptu:
Zaczynam
Element: 1
Element: 2
Element: 3
Element: 4
Koncze
```

Pytania

1. Jakie znasz typy danych w języku Python?
2. W jaki sposób dokonywać konwersji pomiędzy poszczególnymi typami?
3. Jakie są różnice pomiędzy krotką a listą?
4. Na czym polega *slicing*?
5. Jak utworzyć pusty słownik?
6. Jak utworzyć pusty zbiór?
7. Jak działa instrukcja warunkowa?
8. Co oznacza wartość None?
9. Jak działa pętla `for`?
10. Jak działa pętla `while`?
11. W jaki sposób stworzyć tablicę dwuwymiarową?

Literatura

1. Podstawowe typy danych, https://docs.python.org/3/tutorial/introduction.html
2. Struktury danych (Kolekcje), https://docs.python.org/3/tutorial/datastructures.html
3. Instrukcje sterujące, https://docs.python.org/3/tutorial/controlflow.html