Podstawy programowania i przetwarzania danych

Operatory, pliki, reprezentacja typów danych w pamięci komputera

Małgorzata Śleszyńska-Nowak
malgorzata.nowak@pw.edu.pl
http://pages.mini.pw.edu.pl/~sleszynskam/

Wydział Matematyki i Nauk Informacyjnych
Politechnika Warszawska

Spis treści

Operatory

Operatory w języku Python

Operatory arytmetyczne

Według priorytetu (od najwyższego do najniższego)

- `**` (potęgowanie)
- `+`,`-` (unarny plus i minus)
- `*`, `/`, `//`, `%` (mnożenie, dzielenie, dzielenie całkowitoliczbowe, dzielenie modulo)
- `+`,`-` (dodawanie, odejmowanie)

z liczb w liczby

Unarny minus

Minus przed wyrażeniem to osobny operator, tzw. minus unarny (zmieniający znak wyrażenia na przeciwny)

```
1  b = -5
2  b = -b
3  # jaka jest wartość b?
```

Unarny minus

Minus przed wyrażeniem to osobny operator, tzw. minus unarny (zmieniający znak wyrażenia na przeciwny)

```
1  b = -5
2  b = -b
3  # jaka jest wartość b?  5
```

Operatory przykład

Przykład

```
1  a = -3 ** 2              # -9
2  a = (-3) ** 2            # 9
3  a = -3 * -3              # 9
4  a = -5 + 4 * -2 ** 2     # ?
```

Operatory przykład

Przykład

```
1  a = -3 ** 2              # -9
2  a = (-3) ** 2            # 9
3  a = -3 * -3              # 9
4  a = -5 + 4 * -2 ** 2     # ?  -21
```

Operatory w języku Python

Operatory relacyjne

Wszystkie mają ten sam priorytet (niższy niż operatory arytmetyczne)

- `<` (mniejsze)
- `<=` (mniejsze równe)
- `>` (większe)
- `>=` (większe równe)
- `==` (równe)
- `!=` (różne)

z liczb w logiczne

Operatory w języku Python

Operatory logiczne

Według priorytetu (od najwyższego do najniższego), mają niższy priorytet niż operatory relacyjne

- `not` (nie)
- `and` (i)
- `or` (lub)

z logicznych w logiczne

Operatory przykład

Przykład

```
1  if (a >= 0 and a <= 10) or (a >= 20 and a <=30):
2      ...
3  if a >= 0 and a <= 10 or a >= 20 and a <=30:
4      ...
5  if 0 <= a <= 10 or 20 <= a <= 30:    # Python
6      ...
```

Łańcuch porównań

Możemy dokonać dwóch porównań jednocześnie. Wtedy otrzymujemy wartość `True` wtedy i tylko wtedy, gdy obie nierówności są spełnione jednocześnie.

```
a = 5
3 < a <= 5
# True
```

jest to tożsame z:

```
a = 5
3 < a and a <= 5
# True
```

Operatory łączone (ang. *augmented assignments*)

Czasami chcemy zmodyfikować dotychczasową wartość zmiennej, w szczególności popularnym scenariuszem jest zwiększenie tej wartości o 1:

```
1  a = 5
2  b = 2
3  a += 4
4  b *= -1
5  # jaka jest wartość a i b w tym miejscu?
```

Operatory łączone (ang. *augmented assignments*)

Czasami chcemy zmodyfikować dotychczasową wartość zmiennej, w szczególności popularnym scenariuszem jest zwiększenie tej wartości o 1:

```
1  a = 5
2  b = 2
3  a += 4
4  b *= -1
5  # jaka jest wartość a i b w tym miejscu?  9, -2
```

To samo dla `-=`, `/=`, `//=`, `%=`, `**=`.

Konwersja do ogólniejszego typu

Gdy dokonujemy typowej operacji matematycznej, np. dodawania, pomiędzy dwoma typami liczbowymi, np. `int` i `float`, ale też `float` i `complex`, to wynik jest w typie ogólniejszym:

```
2+3, 2.0+3, 2+3.0, 2.0+3.0
# (5, 5.0, 5.0, 5.0)
```

```
2.0+3.0, 2.0+(3+0j)
# (5.0, (5+0j))
```

Obsługa plików

Obsługa plików

Do tej pory wszystkie dane wczytywaliśmy z klawiatury i wypisywaliśmy na konsolę.

W praktyce w większości programów do wczytania i zapisywania danych używa się *plików*.

Do stworzenia obiektu pliku Pythona służy funkcja `open()`.

```
1  plik = open("plik.txt")
2  # lub with open("plik.txt") as plik - będzie dalej
```

Do funkcji `open()` przekazujemy nazwę pliku (albo ścieżkę) i tryb otwarcia (domyślnie "r").

Otwieranie plików

Tryby otwarcia:

- "r" – tylko odczyt z pliku, plik musi istnieć
- "w" – tylko zapis do pliku, jeżeli plik nie istnieje to zostanie utworzony, jeżeli istnieje to jego zawartość zostanie nadpisana
- "a" – tylko dopisanie do pliku, jeżeli plik nie istnieje to zostanie utworzony, jeżeli istnieje to będziemy dopisywać na końcu
- "r+" – odczyt i zapis do pliku, plik musi istnieć
- "w+" – odczyt i zapis do pliku, jeżeli plik nie istnieje to zostanie utworzony, jeżeli istnieje to jego zawartość zostanie nadpisana
- "a+" – odczyt i zapis do pliku, jeżeli plik nie istnieje to zostanie utworzony, jeżeli istnieje to będziemy dopisywać na końcu

Praca z plikami

Otwierając plik za pomocą funkcji `open()` musimy go ręcznie zamknąć.

```
plik = open("plik.txt", "r+")
# praca z plikiem
plik.close()
```

Możemy także otwierać plik za pomocą `with open()`, wówczas zamykany jest automatycznie.

```
with open("plik.txt", "r+") as plik:
    # praca z plikiem
    # plik jest otwarty w trakcie wykonywania tych linii
# a tu już jest zamknięty
```

Wczytywanie treści z pliku

Wczytanie całego pliku do jednego napisu

```
tresc = plik.read()
```

Wczytanie jednej linii

```
linia = plik.readline()
```

Wczytanie całego pliku linia po linii

```
for linia in plik:
    print(linia, end = "")
# czytamy wszystkie linie po kolei, aż do końca pliku
# linie czytane są wraz ze znakiem końca wiersza
```

Zapisywanie do pliku

```
1  plik.write("Zapisujemy ten tekst do pliku\n")
2  plik.write("Write nie dodaje znaku konca linii, ")
3  plik.write("sami musimy go pilnowac.\n")
4  plik.write("Mozemy formatowac tak jak w funkcji print:\n")
5  x = 100
6  plik.write(f"Wartosc zmiennej x = {x}.")
```

*Materiał dodatkowy

```python
plik.write(f"Formatowanie: {2.3456:<15.4f}\n")   # wyrównanie do lewej
plik.write(f"Formatowanie: {2.3456:>15.3f}\n")   # wyrównanie do prawej
plik.write(f"Formatowanie: {2.3456:^15.2f}\n")   # wyśrodkowanie
plik.write(f"Formatowanie: {2.3456:>15.0f}\n")   # f - float
plik.write(f"Formatowanie: {2:>15d}\n")          # d - int
```

```
Formatowanie: 2.3456
Formatowanie:           2.346
Formatowanie:      2.35
Formatowanie:               2
Formatowanie:               2
```

*Materiał dodatkowy

```
1  plik.write(f"Tekst {'drugi tekst':^20s} trzeci tekst\n")
2  plik.write(f'Tekst {"drugi tekst":^20s} trzeci tekst\n')
3  print("Mozemy tez zapisywac z uzyciem print'a", file = plik)
```

```
Tekst      drugi tekst       trzeci tekst
Tekst      drugi tekst       trzeci tekst
Mozemy tez zapisywac z uzyciem print'a
```

Reprezentacja typów danych w pamięci komputera

Liczby całkowite nieujemne

Rozważania zaczniemy od zapisu liczb całkowitych nieujemnych (*unsigned*), czyli zbioru $\mathbb{N}$.

- Przyzwyczajeni jesteśmy do liczb w dziesiętnym systemie pozycyjnym
- „Dziesiętny" oznacza, że mamy cyfry od 0 do 9, a liczbę 10 podnosimy do kolejnych potęg
- „Pozycyjny" oznacza, że pozycja cyfry oznacza rząd wielkości (jedności, dziesiątki, setki...)
- Najmniej znacząca cyfra jest po prawej
- Np. przykładowo, liczba 567 to:
  $5 \times 100 + 6 \times 10 + 7 \times 1 = 5 \times 10^2 + 6 \times 10^1 + 7 \times 10^0$

Liczby całkowite nieujemne

Bardziej ogólnie mamy $(m+1)$-cyfrową liczbę $d_m d_{m-1} \ldots d_1 d_0$, której wartość może być obliczona przy użyciu wzoru:

$$\sum_{i=0}^{m} d_i 10^i$$

- d_m jest cyfrą najbardziej znaczącą, a d_0 – najmniej
- Używanie bazy, czyli liczby 10, jest ogólnie przyjętą normą, ale równie dobrze można wybrać inną liczbę
- W komputerach używamy bazy 2, jako że łatwo jest technicznie mieć dwa stany: *jest* (prąd, wartość 1) i *nie ma* (prądu, wartość 0)

Liczby całkowite nieujemne

- `0b10101` – $10101_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 16 + 4 + 1 = 21$
- `bin(21) == '0b10101'`

Liczby całkowite

- W komputerach mamy ustaloną liczbę bitów, na których zapisujemy liczbę całkowitą
- Najczęściej używamy dwóch formatów:
- 32 bity (4 bajty): $\{-2^{31}, \ldots, 2^{31} - 1\} = \{-2147483648, 2147483647\}$
- 64 bity (8 bajtów): $\{-2^{63}, \ldots, 2^{63} - 1\} = \ldots$

Reprezentacja liczb ze znakiem

- kiedyś używano pierwszego bitu, aby zakodować znak liczby (jeśli był równy 1, to była ona ujemna). Niestety, w takim kodowaniu wykonywanie działań arytmetycznych (dodawania, odejmowania) jest skomplikowane. Dodatkowym problemem są dwa sposoby zapisu liczby zero (tzw. zero ujemne i dodatnie). W dzisiejszych czasach nikt nie używa tego kodowania.
- Używa się za to kodowania U2, w którym łatwo się dodaje i odejmuje liczby, a także jest jedno zero. Liczbę $(d_m d_{m-1} \dots d_1 d_0)_2$ koduje się tak:

$$-d_m 2^m + \sum_{i=0}^{m-1} d_i 2^i$$

Liczby zmiennopozycyjne

- Pewnym podejściem jest podejście stałopozycyjne:
  $\pm$ stały rozmiar liczby całkowitej $\times$ stała skala
- Np. liczba 1234 przy skali 10^{-3} to $1,234$.
- Problemy z kodowaniem ułamków, np. $1/3 = 0.(3)$
- Problem z dodawaniem:
  $1/3 + 1/3 \to 0.333 + 0.333 \to 0.666 < 0.667 \leftarrow 2/3$
- a co z liczbami bardzo małymi? bardzo dużymi?

Liczby zmiennopozycyjne

- Kodowanie zmiennopozycyjne: $(-1)^{\text{znak}}$ mantysa $\times 2^{\text{wykladnik}}$
- Istnieją dwie wersje: 32-bitowa (pojedyncza, *single*) i 64-bitowa (podwójna precyzja, *double precision*)
- Ze względu na bazę 2, dokładne przechowywanie ułamków takich jak 1/10, 1/3 czy 1/5 jest niemożliwe.
- Suma bardzo dużej i bardzo małej liczby da nam w wyniku liczbę bardzo dużą (nie uwzględnimy małej): `1e100+1e-100 == 1e100` zwróci prawdę.

Napisy

- Napisy to ciągi znaków
- Każdy znak może być zakodowany przez stałą lub zmienną liczbę bajtów
- American Standard Code for Information Interchange (US-ASCII) używa jednego bajtu na znak, ale tak naprawdę używa tylko wartości $00 - 7F_{16}$
- Lista kodów jest dostępna np. pod adresem `http://www.asciitable.com/`

```
1 chr(65)
2 # 'A'
3 ord("A")
4 # 65
```

Napisy

- W zglobalizowanym świecie internetu potrzebujemy więcej, niż 128 znaków
- Odpowiedzią jest *The Unicode family* – Universal Coded Character Set
- Dostępne kodowania:
  - UTF-8 – zmienna liczba bajtów na znak
  - UTF-16 – 16 lub 32 bity na znak (1 bajt to 8 bitów)
  - UTF-32 – 32 bity na znak
- Ponad 92% wszystkich stron internetowych używa UTF-8.

- UTF-8 jest kodowaniem o zmiennej liczbie bajtów (od 1 do 4), wstecznie kompatybilnym z ASCII
- Najczęściej używane znaki (we wszystkich współczesnych językach) i symbole (np. interpunkcja, znaki walut, strzałki, operatory matematyczne, kształty geometryczne, alfabet Braille'a) są w płaszczyźnie Unicode 0 (*Unicode plane 0, Basic Multilingual Plane, BMP*) – pierwsze 65 536 znaków 0x00 do 0xFFFF.

Liczba bajtów	Wolne bity	Start	Koniec	Bajt 1	Bajt 2	Bajt 3	Bajt 4
1	7	0x00	0x7F	0xxxxxxx			
2	11	0x80	0x7FF	110xxxxx	10xxxxxx		
3	16	0x800	0xFFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	0x10000	0x1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Wady i zalety UTF-8:

- \+ poprawne ASCII to poprawny UTF-8
- \+ pełny Unicode, znaki wszystkich języków są obsługiwane (co najwyżej użytkownicy mogą nie mieć zainstalowanych odpowiednich czcionek)
- \+ oszczędza miejsce w pamięci, szczególnie dla zachodnich języków
- \+ łatwy do wykrycia, probabilistycznie, jasne wzorce (bajty kontynuacyjne mają prefiks 10_2 itp.), nie wszystkie sekwencje są poprawnym UTF-8
- − niemożliwy losowy dostęp do i-tego znaku w napisie
- − długość napisu nie jest równa liczbie bajtów

Dziękuję

m.sleszynska@mini.pw.edu.pl