rozdział 3:

OPERATORY

ostatnia modyfikacja: 04.12.18

Podstawy programowania

definicje:

- **operator** to symbol, który oddziaływuje na wartość danych (wykonuje pewną operację)
- dana, na którą działa pewien operator, nazywana jest **argumentem** tego operatora
- argumenty wraz z operatorem/operatorami tworzą **wyrażenie**

Podstawy programowania

operatory numeryczne:

- operatory, które zakładają, że ich argumenty są liczbami
- w większości działają zgodnie z naszą arytmetyczną intuicją
- wyjątki – na szczęście – są nieliczne

Podstawy programowania

operator dodawania: + (plus)

- oblicza sumę swojego lewego i prawego argumentu
- np. `3 + 7.5` → `10.5`

Podstawy programowania

operator odejmowania: – (minus)

- oblicza różnicę swojego lewego i prawego argumentu
- np. `3 - 7.5` → `-4.5`

Podstawy programowania

operator mnożenia: * (gwiazdka)

- oblicza iloczyn swojego lewego i prawego argumentu
- np. `3 * 7.5` → `22.5`

Podstawy programowania

operator dzielenia: / (ukośnik)

- oblicza iloraz swojego lewego i prawego argumentu
- np. `3 / 7.5` → `0.4`

Podstawy programowania

operator dzielenia całkowitego: // (ukośnik-ukośnik)

- oblicza **całkowity** iloraz swojego lewego i prawego argumentu
- innymi słowy, wynik zawsze jest liczbą całkowitą – w normalnym dzieleniu tak nie jest – czyli:

`10 // 4` → `2`

ale:

`10 / 4` → `2.5`

Podstawy programowania

operator modulo: `%` (procent)

- oblicza resztę z dzielenia swojego lewego argumentu przez prawy argument
- np: `10 % 3` → `1`

Podstawy programowania

operator potęgowania: ** (gwiazdka-gwiazdka)

- oblicza wartość lewego argumentu podniesionego do potęgi wyrażonej przez prawy argument
- np:

```
2 ** 8 → 256
9 ** 0.5 → 3.0
```

Podstawy programowania

ważne:

- rzeczy **zabronione** w tradycyjnej arytmetyce są też **zabronione** w Pythonie, więc:
- nie próbuj dzielić przez zero
- nie próbuj pierwiastkować liczby ujemnej
- etc...

Podstawy programowania

ważne:

- operatory stosują się do **priorytetów**, tzn., że jedne z nich wykonują swoje operacje przed innymi (jak w tradycyjnej arytmetyce)
- operatory o identycznym priorytecie obliczane są od lewej do prawej

Podstawy programowania

priorytety operatorów (od najwyższego do najniższego)

- **
- * / % //
- + -

Podstawy programowania

nawiasy:

- naturalną kolejność obliczeń można zmienić stosując nawiasy
- używamy wyłącznie nawiasów okrągłych (i)
- pary nawiasów można dowolnie w sobie zanurzać, np. tak:
  ((((((((2+2))))))))
- edytor IDLE będzie nam w tym pomagał

Podstawy programowania

operatory bitowe:

- operatory przedstawione wcześniej działają jednakowo skutecznie na liczbach całkowitych i rzeczywistych
- oprócz nich istnieją operatory działające tylko na liczbach całkowitych
- są to tzw. operatory **bitowe**, bo oddziałują na każdy bit dane z osobna

Podstawy programowania

funkcja bin():

- przyda się do eksperymentów z operatorami bitowymi
- `bin(x)`:
  - argument: liczba całkowita
  - wynik: postać binarna argumentu
  - efekt: żaden
- np:
- `print(bin(15))` → `0b1111`

Podstawy programowania

funkcja hex():

- `hex(x)`:
  - argument: liczba całkowita
  - wynik: postać szesnastkowa argumentu
  - efekt: żaden
- np:
- `print(hex(15))` → `0xf`

Podstawy programowania

operator negacji bitowej: ~ (tylda)

- działanie (tabela prawdy):

x	~x
0	1
1	0

Podstawy programowania

operator negacji bitowej: ~

- np:
  `print(bin(~0b101)) → -0b110`
- czemu tak?

Podstawy programowania

operator iloczynu bitowego: & (ampersand)

- działanie (tabela prawdy):

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Podstawy programowania

operator iloczynu bitowego: &

- np:
  `print(bin(0b110 & 0b011)) → 0b10`

- zauważ:
  `1 & x → x`
  `0 & x → 0`

Podstawy programowania

operator sumy bitowej: | (bar)

- działanie (tabela prawdy):

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Podstawy programowania

operator sumy bitowej: |

- np:
  `print(bin(0b110 | 0b011)) → 0b111`

- zauważ:
  `1 | x → 1`
  `0 | x → x`

Podstawy programowania

operator sumy wykluczające (xor): ^ (caret)

xor: ang. exclusive or

- działanie (tabela prawdy):

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Podstawy programowania

operator xor: ^

- np:
  `print(bin(0b110 ^ 0b011)) → 0b101`

- zauważ:
  `1 ^ x → ~x`
  `0 ^ x → x`

Podstawy programowania

operator przesuwania bitowego w lewo: <<

```
a << b
```

- przesuń bity w *a* o *b* pozycji w lewo
- na puste miejsca wprowadź bity 0

- np:

```
print(bin(0b110 << 2)) → 0b11000
print(0b110 << 2)      → 24
```

- zauważ:

```
x << 1 → 2 * x
```

Podstawy programowania

operator przesuwania bitowego w prawo: >>

```
a >> b
```

- przesuń bity w *a* o *b* pozycji w prawo
- bity uciekające z danej giną bez śladu :(

- np:

```
print(bin(0b110 >> 1)) → 0b11
print(0b110 >> 1)      → 3
```

- zauważ:

```
x >> 1 → x // 2
```

Podstawy programowania

operatory relacyjne:

- operatory relacyjne stwierdzają w jakiej wzajemnej **relacji** pozostają ich argumenty
- innymi słowy: **porównują je**
- wynikiem ich działania jest zawsze prawda (True) albo fałsz (False)

Podstawy programowania

operator > (większe)

- sprawdza, czy lewy argument jest większy od prawego
- np.

```
3 > 7.5 → False
7.5 > 3 → True
3 > 3   → False
```

Podstawy programowania

operator >= (większe równe)

- sprawdza, czy lewy argument jest większy od prawego lub mu równy
- np.

```
3 >= 7.5  → False
7.5 >= 3  → True
3 >= 3    → True
```

Podstawy programowania

operator < (mniejsze)

- sprawdza, czy lewy argument jest mniejszy od prawego
- np.

```
3 < 7.5 → True
7.5 < 3 → False
3 < 3   → False
```

Podstawy programowania

operator <= (mniejsze równe)

- sprawdza, czy lewy argument jest mniejszy od prawego lub mu równy
- np.

```
3 <= 7.5  → True
7.5 <= 3  → False
3 >= 3    → True
```

Podstawy programowania

operator == (równe równe)

- sprawdza, czy lewy argument **jest równy** prawemu
- np.

```
3 == 7.5  → False
7.5 == 3  → False
3 == 3    → True
```

Podstawy programowania

zapamiętaj!

- do porównywania danych **używaj** operatora ==
- operator = służy do czegoś zupełnie **innego**!
- nie myl ich, bo sprowadzisz na siebie **kłopoty** :)

Podstawy programowania

operator != (różne)

- sprawdza, czy lewy argument **nie jest równy** prawemu
- np.

```
3 != 7.5  → True
3 != 3    → False
```

Podstawy programowania

operatory logiczne:

- operatory logiczne działają na wartościach logicznych, łącząc je w złożone **zdania**
- ich działaniem rządzi tzw. **algebra Boole'a**
- z tego też powodu nazywa się je czasem operatorami **boolowskimi**

Podstawy programowania

operator zaprzeczenia: `not` (nie)

- działanie (tabela prawdy):

x	not x
False	True
True	False

Podstawy programowania

operator zaprzeczenia: not

- np:

```
print(1 > 2)        → False
print(not 1 > 2)    → True
```

Podstawy programowania

operator koniunkcji: and (i)

- działanie (tabela prawdy):

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

Podstawy programowania

operator koniunkcji: and

- np:

```
print(1 > 0 and 0 < 1) → True
print(1 > 0 and 0 > 1) → False
```

Podstawy programowania

operator alternatywy: `or` (lub)

- działanie (tabela prawdy):

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Podstawy programowania

operator koniunkcji: or

- np:

```
print(1 < 0 or 0 > 1)  → False
print(1 > 0 or 0 > 1)  → True
```

Podstawy programowania

zauważ:

- `not 1 < 0 → 1 >= 0`
  `not 1 > 0 → 1 <= 0`

Małe repetytorium: prawa de Morgana:

- *zaprzeczenie alternatywy jest równe koniunkcji zaprzeczeń*
- *zaprzeczenie koniunkcji jest równe alternatywie zaprzeczeń*

- sprawdźmy, czy to działa...

Podstawy programowania

prawo 1:

```
print(not (1 < 0 or 1 > 0))
→ False

print((not 1 < 0) and (not 1 > 0))
→ False

print(1 >= 0 and 1 <= 0)
→ False
```

Podstawy programowania

prawo 2:

```
print(not (1 < 0 and 1 > 0))
→ True

print((not 1 < 0) or (not 1 > 0))
→ False

print(1 >= 0 or 1 <= 0)
→ False
```

konkluzja: prawa de Morgana działają

możemy spać spokojnie ;)