Úvod do jazyka Python a počítačového programování

Modul 3 sekce 3 - Logické a bitové operace v jazyce Python

V této části se seznámíte s logickými a bitovými operátory v jazyce Python a s pojmy jako pravdivostní tabulka a bitový posun.

3.1 Počítačová logika

Všimli jste si, že podmínky, které jsme dosud používali, byly velmi jednoduché, neřkuli primitivní? Podmínky, které používáme v reálném životě, jsou mnohem složitější. Podívejme se na tuto větu:

*Pokud budeme mít trochu volného času **a** bude pěkné počasí, půjdeme se projít.*

Použili jsme spojku ***a***, což znamená, že jít na procházku závisí na současném splnění těchto dvou podmínek. V jazyce logiky se takové spojení podmínek nazývá konjunkce. A nyní další příklad:

*Pokud jsi v obchodním centru ty **nebo** já, jeden z nás koupí dárek pro maminku.*

Výskyt slova ***nebo*** znamená, že nákup závisí alespoň na jedné z těchto podmínek. V logice se takové spojení nazývá disjunkce.

Je jasné, že Python musí mít operátory pro sestavování konjunkcí a disjunkcí. Bez nich by byla vyjadřovací síla jazyka značně oslabena. Říká se jim **logické operátory**.

Operátor `and`

Jedním z operátorů logické konjunkce v jazyce Python je slovo `and`. Je to binární operátor s prioritou, která je nižší než priorita vyjádřená operátory porovnávání. Umožňuje nám kódovat složité podmínky bez použití závorek, jako je tato:

```
counter > 0 and value == 100
```

Výsledek poskytnutý operátorem `and` lze určit na základě pravdivostní tabulky.

Uvažujeme-li konjunkci `A a B`, vypadá množina možných hodnot argumentů a odpovídajících hodnot konjunkce následovně:

Argument A	Argument B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Operátor `or`

Operátor disjunkce je slovo `or`. Je to binární operátor s **nižší prioritou než operátor** `and` (stejně jako `+` oproti `*`). Jeho pravdivostní tabulka je následující:

Argument `A`	Argument `B`	`A or B`
False	False	False
False	True	True
True	False	True
True	True	True

Operátor `not`

Kromě toho existuje ještě jeden operátor, který lze použít při konstrukci podmínek. Jedná se o unární operátor provádějící logickou negaci. Jeho operace je jednoduchá: mění pravdu na nepravdu a nepravdu na pravdu.

Tento operátor se zapisuje jako slovo `not` a **jeho priorita je velmi vysoká**: stejná jako u unárních operátorů `+` a `−`. Jeho pravdivostní tabulka je jednoduchá:

Argument	`not` Argument
False	True
True	False

3.2 Logické výrazy

Vytvořme proměnnou s názvem `var` a přiřaďme jí hodnotu `1`. Následující podmínky jsou párově ekvivalentní:

```
# Example 1:
print(var > 0)
print(not (var <= 0))

# Example 2:
print(var != 0)
print(not (var == 0))
```

Možná znáte De Morganovy zákony. Ty říkají, že:

Negace konjunkce je disjunkcí negací.

Negace disjunkce je konjunkce negací.

Napišme totéž pomocí jazyka Python:

```
not (p and q) == (not p) or (not q)
not (p or q) == (not p) and (not q)
```

Všimněte si, jak jsou v kódu výrazů použity závorky - vložili jsme je tam kvůli lepší čitelnosti.

3.3 Logické hodnoty vs. jednotlivé bity

Logické operátory berou své argumenty jako celek bez ohledu na to, kolik bitů obsahují. Operátory si uvědomují pouze hodnotu: nula (když jsou všechny bity vynulovány) znamená `False`; nenulová hodnota (když je nastaven alespoň jeden bit) znamená `True`.

Výsledkem jejich operací je jedna z těchto hodnot: `False` nebo `True`. To znamená, že následující úryvek přiřadí proměnné `j` hodnotu `True`, pokud `i` není nulové; v opačném případě bude mít hodnotu `False`.

```
i = 1
j = not not i
```

3.4 Bitové operátory

Existují však čtyři operátory, které umožňují manipulovat s jednotlivými bity dat. Nazývají se bitové operátory.

Zahrnují všechny operace, o kterých jsme se zmínili dříve v logickém kontextu, a jeden operátor navíc. Jedná se o operátor `xor` (jako exkluzivní nebo), který se označuje jako ^ (caret).

Zde jsou všechny:

& (ampersand) - bitová konjunkce; (alt 38)

| (bar) - bitová disjunkce; (alt 124)

~ (tilda) - bitová negace; (alt 126)

^ (caret) - bitová exkluzivita nebo (xor); (alt 94)

Bitwise operations (&, |, and ^)				
Argument `A`	**Argument `B`**	**`A & B`**	**`A | B`**	**`A ^ B`**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise operations (~)	
Argument	**~ Argument**
0	1
1	0

Usnadníme vám to:

`&` vyžaduje přesně dvě jedničky, aby výsledkem byla jednička;

`|` vyžaduje alespoň jednu jedničku, aby výsledkem byla jednička;

`^` vyžaduje přesně jednu jedničku, aby výsledkem byla jednička.

Dodejme důležitou poznámku: argumenty těchto operátorů musí být celá čísla; nesmíme zde používat floaty.

Důležitý je rozdíl ve fungování logických a bitových operátorů: logické operátory nepronikají do bitové úrovně svého argumentu. Zajímá je pouze výsledná celočíselná hodnota.

Bitové operátory jsou přísnější: zabývají se každým bitem zvlášť. Předpokládáme-li, že celočíselná proměnná zabírá 64 bitů (což je v moderních počítačových systémech běžné), můžete si bitovou operaci představit jako 64násobné vyhodnocení logického operátoru pro každou dvojici bitů argumentů. Tato analogie je samozřejmě nedokonalá, protože v reálném světě se všech těchto 64 operací provádí současně (simultánně).

Logické vs. bitové operace

Nyní si ukážeme příklad rozdílu mezi logickými a bitovými operacemi. Předpokládejme, že byla provedena následující přiřazení:

```
i = 15
j = 22
```

Předpokládáme-li, že celá čísla jsou uložena s 32 bity, bude bitový obraz obou proměnných následující:

```
i: 00000000000000000000000000001111
j: 00000000000000000000000000010110
```

Úkol je zadán:

```
log = i and j
```

Máme zde co do činění s logickou spojkou. Sledujme průběh výpočtů. Obě proměnné `i` a `j` nejsou nulové, takže je budeme považovat za reprezentanty `True`. Nahlédnutím do pravdivostní tabulky pro operátor `and` vidíme, že výsledkem bude `True`. Žádné další operace se neprovádějí.

```
log: True
```

Nyní bitová operace - zde je:

```
bit = i & j
```

Operátor `&` bude pracovat s každou dvojicí odpovídajících bitů zvlášť a vytvoří hodnoty příslušných bitů výsledku. Výsledek tedy bude vypadat takto:

i	01111
j	10110
bit = i & j	00110

Tyto bity odpovídají celočíselné hodnotě šest. Podívejme se nyní na operátory negace. Nejprve logický:

```
logneg = not i
```

Proměnná logneg bude nastavena na hodnotu False - nic dalšího není třeba dělat. Bitová negace probíhá takto:

```
bitneg = ~i
```

To může být trochu překvapivé: hodnota proměnné bitneg je -16. Může se to zdát divné, ale vůbec to tak není. Pokud se chcete dozvědět více, měli byste se seznámit s binární číselnou soustavou a pravidly pro zápis dvojkových čísel.

i	00000000000000000000000000001111
bitneg = ~i	11111111111111111111111111110000

Každý z těchto dvouargumentových operátorů lze použít ve zkrácené podobě. Toto jsou příklady jejich ekvivalentních zápisů:

x = x & y	x &= y
x = x | y	x |= y
x = x ^ y	x ^= y

3.5 Jak se vypořádáme s jednotlivými bity?

Nyní si ukážeme, k čemu všemu lze bitové operátory použít. Představte si, že jste vývojář, který musí napsat důležitou součást operačního systému. Bylo vám řečeno, že smíte použít proměnnou přiřazenou následujícím způsobem:

```
flag_register = 0x1234
```

Proměnná uchovává informace o různých aspektech fungování systému. Každý bit proměnné uchovává jednu hodnotu ano/ne. Bylo vám také řečeno, že pouze jeden z těchto bitů je váš - třetí (nezapomeňte, že bity jsou číslovány od nuly a bit číslo nula je nejnižší, zatímco nejvyšší je číslo 31). Zbývající bity se nesmí měnit, protože jsou určeny k ukládání jiných dat. Zde je váš bit označený písmenem x:

```
flag_register = 00000000000000000000000000x000
```

Můžete se setkat s následujícími úkoly:

1. Zjistit stav bitu - chcete zjistit hodnotu bitu; porovnáním celé proměnné s nulou nic nezjistíte, protože zbývající bity mohou mít zcela nepředvídatelné hodnoty, ale můžete použít následující vlastnost konjunkce:

```
x & 1 = x
x & 0 = 0
```

Pokud použijete operaci `&` na proměnnou `flag_register` spolu s následujícím bitovým obrazem:

```
00000000000000000000000000001000
```

(všimněte si jedničky na pozici vašeho bitu), získáte jeden z následujících řetězců bitů:

`00000000000000000000000000001000`, pokud byl váš bit nastaven na `1`

`00000000000000000000000000000000`, pokud byl váš bit resetován na `0`

Taková posloupnost nul a jedniček, jejímž úkolem je zachytit hodnotu nebo změnit vybrané bity, se nazývá bitová maska.

Sestavme si bitovou masku pro zjištění stavu bitu. Měla by ukazovat na třetí bit.

Tento bit má váhu $2^3 = 8$. Vhodnou masku lze vytvořit následující deklarací:

```
the_mask = 8
```

Můžete také vytvořit posloupnost instrukcí v závislosti na stavu bitu. Zde je:

```
if flag_register & the_mask:
    # My bit is set.
else:
    # My bit is reset.
```

2. Resetujte svůj bit - přiřadíte mu nulu, zatímco všechny ostatní bity musí zůstat beze změny; použijme stejnou vlastnost konjunkce jako dříve, ale použijme trochu jinou masku - přesně tak, jak je uvedeno níže:

```
11111111111111111111111111110111
```

Všimněte si, že maska vznikla negací všech bitů proměnné `the_mask`. Nulování bitů je jednoduché a vypadá takto (vyberte si ten, který se vám líbí více):

```
flag_register = flag_register & ~the_mask
flag_register &= ~the_mask
```

3. Nastavte svůj bit - svému bitu přiřadíte jedničku, zatímco všechny ostatní bity musí zůstat beze změny; použijte následující vlastnost disjunkce:

```
x | 1 = 1
x | 0 = x
```

Nyní jste připraveni nastavit bit pomocí jednoho z následujících pokynů:

```
flag_register = flag_register | the_mask
flag_register |= the_mask
```

4. Negace bitu - nahradíte 1 za 0 a 0 za 1. Můžete využít zajímavou vlastnost operátoru `xor`:

```
x ^ 1 = ~x
x ^ 0 = x
```

a negujte svůj bit pomocí následujících pokynů:

```
flag_register = flag_register ^ the_mask
flag_register ^= the_mask
```

Odkazy:

Cisco Programming Essentials in Python

Root.cz

ITNetwork.cz

Internet

https://cs.from-locals.com/python-bit-operation/

https://www.builder.cz/rubriky/c/c--/ucime-se-c-25-dil-bitove-operatory-a-bitove-pole-155778cz