

02 zanimivejši pogoji

January 28, 2024

0.1 Logični operatorji ne delajo le z bool

V Pythonu seveda velja, tako kot v Cju, da je 0 neresnična, vsa druga števila pa so resnična.

```
[ ]: >>> 0 or 1
1
>>> 0 and 0
0
```

Prav tako veljajo za neresnične `None` (`None` je malo podoben Cjevskumu `NULLu`, vendar ne preveč, saj Python nima kazalcev), neresničen je tudi prazen niz, prazen seznam... vse, kar je prazno. Do sem nič posebnega in nepričakovanega.

Posebno pa je tole: človek bi pričakoval, da `and` vrne `True`, če sta leva in desna stran resnični, in `False`, če ne. Prav tako bi pričakoval, da bo `or` vrnil `True`, če je vsaj ena od strani resnična in `False`, če ni. No, tak človek bi se motil.

Pravilo je takšno:

- **and**: če je vrednost na levi strani neresnična, jo vrne; sicer vrne vrednost na desni.
- **or**: če je vrednost na levi strani resnična, jo vrne; sicer vrne vrednost na desni.

Na prvi pogled tidve pravili samo na malo bolj zapleten način opisujeta, kar pač morata početi `and` in `or`. Da, že, vendar v primeru, da imamo samo `True` in `False`. `False and <karkoli>` vrne levo stran, torej `False`; `True and False` vrne desno stran, torej `False`; `True and True` vrne desno stran, torej `True`.

Ker je tudi za vsako drugo stvar določeno, ali je resnična ali neresnična (kot smo napisali: število je resnično, če je različno od 0; različne druge stvari so resnične, če so neprazne), pa imata pravili zanimive posledice.

```
[ ]: >>> 1 and 1
1
>>> 1 and 2
2
>>> 2 and 1
1
>>> 2 and 0
0
>>> 0 and 2
0
```

```
>>> "" and 2
''
>>> 3 or 0
3
>>> 3 or ""
3
>>> 2 and 3 or 4
3
>>> 2 and 3 or 0
3
>>> 2 and 3 or 4
3
```

Se zdi uporabno? Ne? Pa pogledjte tole:

```
[ ]: >>> '' or '<neregistriran uporabnik>'
'<neregistriran uporabnik>'
>>> 'Janez' or '<neregistriran uporabnik>'
'Janez'
```

Kadar programiram v kakem drugem jeziku, ne Pythonu, tole res pogrešam.

0.2 if-else AKA Pythonov japonski operator

Kaj pa naredi tole?

```
[2]: x = 5

x > 0 and 42 or 13
```

[2]: 42

Ali pa, recimo, tako (da bo počelo kaj smiselnega):

```
[3]: x > 0 and x or 0
```

[3]: 5

Branje moramo začeti pri `or`, navsezadnje je izraz, v bistvu, tak: `(x > 0 and 42) or 13`. Zdaj najprej pogledamo levo stran. Tam imamo `and`. - Če je `x` več kot 0, bo rezultat izraza `x > 0 and 42` enak 42, torej imamo `42 or 13`, to pa je 42. - Če `x` ni večji od 0, pa bo `and` neresničen, `False`, torej imamo `False or 13`, kar je 13.

Ali pa takole:

```
[9]: x = 5
y = 8

x > y and x or y
```

[9]: 8

Če je x večji od y , je rezultat x , sicer y .

Nas to na kaj spominja? No, da to je isto kot $x > y ? x : y$. Vsaj za tiste, ki iz kakega drugega jezika poznate operator $?:$. Iz Pythona ga namreč ne morete, ker ga nima.

V večini jezikov, torej, imamo “ternarni” operator $?:$. Ternarni? No, da. Operatorji so lahko unarni, tako kot, na primer unarni minus (-3), `not` (`not b` ali, v C-jevski sintaksi, `!b`) ali dvojiški komplement, `~`. Lahko so binarni, kot `+`, `-`, `*`, `/` in tako naprej; to so tisti, ki sprejmejo dva operanda. Lahko pa so ternarni, kot je $?:$.

$?:$ sprejme tri operande. Izraz $a ? b : c$ ima vrednost b , kadar je a resničen, sicer pa c . Ker Python tega nima (oziroma, ni imel in še vedno nima v tej obliki), so programerji namesto $a ? b : c$ pisali `a and b or c`. Kar je večinoma delovalo, včasih pa tudi ne.

```
[10]: x = 0
      y = -5

      x > y and x or y
```

[10]: -5

Zakaj to ne vrne x ? $x > y$ je vendar resničen! Poglejmo, kaj dobimo, če vstavimo x in y (poleg tega pa dodamo še oklepaje, da bo bolj očitno, da se najprej poračuna `and`): `(0 > -5 and 0) or -5` oziroma `(True and 0) or -5`. Ker je 0 neresnično, je to `0 or -5`, torej `-5`.

Debata je bila dolga, trajala je leta. Operatorja $?:$ v Python niso uvedli, ker je grd, nepythonovski. Python je zračen, v njem ni nobenih `&&` in `::*`, torej tudi $?:$: ne bomo tolerirali. Ampak boljšega si pa niso izmislili. Na koncu je benevolentni dosmrtni diktator Pythona odločil, da se bo to zapisovalo z izrazom `if-else`, takole:

```
[11]: x if x > 0 else y
```

[11]: -5

Nek C-ja vaje študent je nekoč komentiral, da je tale operator japonski. Nekako, narobe obrnjen. Ali pa tudi ne: prebere se čisto lepo. Mar nisem zgoraj napisal “Izraz $a ? b : c$ ima vrednost b , kadar je a resničen, sicer pa c .” To je dobesedno to: `b if a else c`.

Vseeno je operator čuden in ga pravzaprav ne uporabljamo radi in pogosto.

Operator `if-else` je del izraza. To ni isto kot `if` in `else`, o katerem smo se sicer poučili danes. Uporabljamo ga lahko, recimo tako:

```
[12]: t = x if x > 0 else y
```

Ali tako

```
[13]: pow(5 * (x if x > 0 else y), 2)
```

[13]: 625

0.3 Ujemanje vzorcev

Tipično vprašanje prišlekov iz drugih jezikov je: a `switch`-a pa Python nima? Za tiste, ki ne veste, kaj je `switch`: v C-ju lahko napišemo

```
switch(err.code) {
    case 400: msg = "Bad request"; break;
    case 404: msg = "Page not found"; break;
    case 418: msg = "I'm a teapot"; break;
    default: msg = "Whatever";
}
```

In podobno v drugih jezikih. V Pythonu pa ne. Ker tam lahko napišeš:

```
if err.code == 400:
    msg = "Bad request"
elif err.code == 404:
    msg = "Page not found"
elif err.code == 418:
    msg = "I'm a teapot"
```

Ali, točneje, ker ti tega ni treba pisati, saj lahko, kot bomo izvedeli čez slab mesec, napišeš

```
messages = {400: "Bad request", 404: "Page not found", 418: "I'm a teapot"}
```

in potem

```
msg = messages[err.code]
```

Tu bi se kdo uprl, da to ni isto: v `switch` lahko dodaš še marsikaj drugega kot en sam izraz. Upor je bil preslišan do različice Pythona 3.10, ki je v tem trenutku stara par dni in spodnje kode sploh še ne morem izvajati v Jupytru, ker ga le-ta še ne podpira. V Pythonu 3.10, torej, lahko napišemo tole:

```
code = 418

match code:
    case 400:
        msg = "Bad request"
    case 404:
        msg = "Page not found"
    case 418:
        msg = "I'm a teapot"
    case _:
        msg = "Whatever"
```

```
print(msg)
```

Na prvi pogled je stvar takšna kot v jezikih, ki so prevzeli C-jevski zapis, samo da so v Pythonu iz navidez neznanega razloga zamenjali `switch` z `match`. To ni nič takega; v Kotlinu, recimo, bi se tole zapisalo kot

```
when (code) {
```

```

400 -> msg = "Bad request";
404 -> msg = "Page not found";
418 -> msg = "I'm a teapot";
else -> msg = "Whatever";
}

```

(Kotlin v resnici carski jezik; gornje se bolj pravilno napiše tako:

```

msg = when (code) {
    400 -> "Bad request";
    404 -> "Page not found";
    418 -> "I'm a teapot";
    else -> "Whatever";
}

```

ker je v Kotlinu **when** pravzprav izraz. Kot tudi večina drugih stvari v Kotlinu.)

No, vrnimo se v Python. Pythonov **match** ni isto kot C-jevski **switch**, temveč veliko več. Pythonov **match** preverja ujemanje strukture, ne le vsebine.

```

match tocka:
    case (0, 0):
        print("izhodišče")
    case (x, y):
        print("ni izhodišče")
    case _:
        print("ni točka")

```

Če ima **tocka** vrednost (0, 0), bo tole izpisalo, "izhodišče"; če ima točka kake druge koordinate, na primer (3, 1), bo izpisalo "ni izhodišče"; če pa ima **tocka** vrednost 42, bo izpisalo "ni točka" - ker pač ni oblike (x, y).

No, to ni še nič. Napišemo lahko tole:

```

match tocka:
    case (0, 0):
        print("izhodišče")
    case (x, 0):
        print("x =", x, "(na osi y)")
    case (0, y):
        print("y =", y, "(na osi x)")
    case (x, y):
        print("tocka, x =", x, "in y =", y)
    case _:
        print("ni točka")

```

Python gre po vrsti in poišče prvi primer, ki opiše **tocka**: najprej preverjamo, ali sta obe koordinati 0, nato ali je druga koordinata enaka 0, nato ali je prva 0, sicer pa, ali sploh imamo dve koordinati. In v vseh teh primerih priredi spremenljivki **x** ali **y** podatek, ki je na tem mestu.

Seveda lahko naredimo tudi to:

```

match tocka:

```

```

case (0, 0):
    print("izhodišče")
case (x, 0):
    print("x =", x, "(na osi y)")
case (0, y):
    print("y =", y, "(na osi x)")
case (x, y):
    print("točka, x =", x, "in y =", y)
case (x, y, z):
    print("olala, tri dimenzije")
case _:
    print("ni točka")

```

No, zakaj bi se ustavili pri treh. Kot bomo videli prihodnji teden (ko bomo tudi boljše razumeli, kaj je tule v ozadju in kaj pomeni tale zvezdica) lahko naredimo tudi to:

```

tocka = (3, 0, 5, 6, 8)
match tocka:
    case (0, 0):
        print("izhodišče")
    case (x, 0):
        print("x =", x, "(na osi y)")
    case (0, y):
        print("y =", y, "(na osi x)")
    case (x, y):
        print("točka, x =", x, "in y =", y)
    case (x, y, *ostale):
        print("olala,", 2 + len(ostale), "dimenzij!")
    case _:
        print("ni točka")

```

Da ne zapletamo brez potrebe z x in y, bi lahko pisali tudi:

```

case (*t, ): print("olala,", len(t), "dimenzij!")

```

Še en lep primer iz [dokumentacije](#):

```

command = "take keys"

match command.split():
    case ("quit", ):
        print("End of game")
    case ("take", object):
        print("You took the", object)
    case ("touch", what):
        print("Don't touch", what)
    case ("talk", "to", person):
        print("You said: 'Hi,", person, "'")

```

Dodati je možno tudi pogoje!

```

tocka = (3, 0)

```

```
match tocka:
    case (0, 0):
        print("izhodišče")
    case (x, 0) if x > 0:
        print("desno od izhodišča, x =", x, "(na osi y)")
    case (x, 0):
        print("levo od izhodišča, x =", x, "(na osi y)")
    case _: print("karkoli")
```

Ko Python najde ujemanje, preveri še pogoj, in če ta ni izpolnjen, nadaljuje s preiskovanjem naslednjih vzorcev.

[]: