09 oblikovanje nizov, datoteke

January 28, 2024

1 Oblikovanje nizov

Kot vsak normalen jezik ima tudi Python nek način oblikovanja nizov. Iz zgodovinskih razlogov pravzaprav tri.

1.0.1 Operator %

V C-ju in jezikih, ki se zgledujejo po njem, imamo vzorce oblike `"Danes sem videl %i primerkov %s."`. To je tudi najstarejša oblika oblikovanja nizov v Pythonu, le da Python nima funkcij, kot je C-jev `printf`, temveč ima niz operator `%` (ja, recikliran operator za ostanek po deljenju), ki mu sledi terka z vrednostmi, ki jih je potrebno vstaviti. Če vstavljamo le eno vrednost, je ni potrebno dajati v terko.

[1]:
```
"Danes sem videl %i primerkov %s." % (7, "medvedov")
```

[1]:
```
'Danes sem videl 7 primerkov medvedov.'
```

[2]:
```
"Danes sem videl %s." % "medveda"
```

[2]:
```
'Danes sem videl medveda.'
```

Seveda lahko vključimo tudi vse ostale C-jevske trike - število mest, decimalk, predznak…

[3]:
```
"  je približno %4.2f." % 3.14159265
```

[3]:
```
'  je približno 3.14.'
```

To starodavno obliko boste v novejši kodi videvali le še v povezavi z modulom `logging`, zato v zvezi z njo ne bomo našteli vsega, kar se da (še) početi z njo in tega (še) ne poznati iz drugih jezikov, kjer ste morda srečali nize v tej obliki. Kdor hoče vedeti, bo pogledal dokumentacijo. Tu pokažimo le še tole: namesto terke lahko podamo slovar.

[4]:
```
"Danes sem videl %(n)i primerkov %(zival)s." % {"n": 7, "zival": "medvedov"}
```

[4]:
```
'Danes sem videl 7 primerkov medvedov.'
```

To je očitno še posebej uporabno predvsem - če je niz zapleten in se ne bi radi izgubljali v vrstnem redu, - ali če imamo tak slovar že ravno pri roki.

Slovar lahko vsebuje tudi odvečne elemente; Python jih bo pač ignoriral.

1.0.2 Metoda `format`

Nizi imajo metodo `format`. Ta znotraj niza poišče vse pare zavitih oklepajev in jih zamenja z istoležnimi argumenti, ki jih podamo ob klicu `format`. Bolj očitno bo iz primera.

[5]:
```
vzorec = "Danes sem videl {} primerkov {}."

vzorec.format(7, "medvedov")
```

[5]:
```
'Danes sem videl 7 primerkov medvedov.'
```

Dele lahko, tako kot prej, tudi poimenujemo.

[6]:
```
"Danes sem videl {n} primerkov {zival}.".format(n=7, zival="medvedov")
```

[6]:
```
'Danes sem videl 7 primerkov medvedov.'
```

Če želimo dodati oblikovanje, dodamo `:` in za njim obliko v običajni obliki.

[7]:
```
"{n:.2f} odstotkov medvedov še ne spi.".format(n=3.14159265)
```

[7]:
```
'3.14 odstotkov medvedov še ne spi.'
```

Če hočemo brez imena, pač naredimo brez.

[8]:
```
"{:.2f} odstotkov medvedov še ne spi.".format(3.14159265)
```

[8]:
```
'3.14 odstotkov medvedov še ne spi.'
```

Ta način je praktičen zato, ker je vzorec shranjen v obliki niza. Kaj s tem mislim, bomo videli, ko izvemo za trenutno aktualen način oblikovanja, kjer ni tako. V praksi pa tudi tega način ne uporabljamo več pogosto, zato bodi tudi o njem dovolj. Berite dokumentacijo.

1.1 Interpolacija

Danes je v vseh sodobnih jezikih v modi veliko prikladnejši način oblikovanja nizov. Tipično ga najdemo pod imenom string interpolation. Python ga ima od različice 3.6.

Ko pišemo niz, pred narekovaj (enojni ali dvojni ali trojni) damo znak `f` in Python bo vse, kar je znotraj zavitih oklepajev izračunal in vstavil v niz.

[9]:
```
n = 7
zival = "medvedov"

f"Danes sem srečal {n} primerkov {zival}."
```

[9]:
```
'Danes sem srečal 7 primerkov medvedov.'
```

Znotraj oklepajev so lahko celi izrazi.

[10]:
```
fahr = 42

f"{fahr} Fahrenheitov je {(fahr - 32) * 5 / 9} Celzijev"
```

[10]:
```
'42 Fahrenheitov je 5.555555555555555 Celzijev'
```

Tako kot pri `format`, lahko izrazu sledi dvopičje in nato format.

[11]:
```
f"{fahr:3.1f} Fahrenheitov je {(fahr - 32) * 5 / 9:3.1f} Celzijev"
```

[11]:
```
'42.0 Fahrenheitov je 5.6 Celzijev'
```

Vse je enako kot v C-ju: v formatu `4.1f` številke, `4.1`, pomenijo, da bi radi izpis na štiri mesta, pri čemer naj bo eno rezervirano za decimalko. Za črko `f` si predstavljajmo, da pomeni `float`. Če pustimo za število premalo prostora, ga bo izpis pač zasedel več.

[12]:
```
x = 1234.5678
f"Primer predolgega števila: {x:3.1f}"
```

[12]:
```
'Primer predolgega števila: 1234.6'
```

Še nekaj oblikovanja:

[13]:
```
podatki = [
    (74, "Anze", False),
    (82, "Benjamin", False),
    (48, "Cilka", True),
    (66, "Dani", False),
    (61, "Eva", True),
    (101, "Franc", False),
    ]

for teza, ime, spol in podatki:
    print(f"{ime:10}{teza:6}")
```

```
Anze          74
Benjamin      82
Cilka         48
Dani          66
Eva           61
Franc        101
```

Znak, ki pove, za kaj gre (število, niz…) lahko tudi izpustimo in Python bo naredil, kot bo prav.

Opazimo, da so nizi privzeto poravnani na levo, številka na desno. To lahko spremenimo, če za dvopičje dodamo "puščico" `<`, `>` ali `^` (centriranje).

[14]:
```
for teza, ime, spol in podatki:
    print(f"{ime:>10} {teza:<6}")
```

```
     Anze 74
 Benjamin 82
    Cilka 48
     Dani 66
      Eva 61
    Franc 101
```

Pred znak, ki določa poravnavo, lahko dodamo simbol, ki naj se uporabi za zapolnjevanje - če seveda nismo zadovoljni s presledkom. Poravnajmo imena levo, števile desno, namesto presledkov pa zapolnimo prazen prostor s pikami.

[15]:
```
for teza, ime, spol in podatki:
    print(f"{ime:.<10}{teza:.>6}")
```

```
Anze…74
Benjamin…82
Cilka…48
Dani…66
Eva…61
Franc…101
```

Namesto pik lahko uporabimo poljuben drug znak. Kadar takole zapolnjujemo prostor s čimerkoli drugim kot s presledki, moramo dodati znake za poravnavanje (`<`, `>` ali `^`), tudi kadar z njimi izberemo takšno poravnavanje, kot bi bilo tudi privzeto (desno za števila, levo za vse drugo) všeč.

Tega je seveda še ogromno. Določimo lahko, naj se vedno izpiše predznak, torej +, kadar je število pozitivno, zahtevamo izpis v dvojiškem ali šestnajstiškem sistemu…. Za števila lahko določimo, naj se izpišejo dvojiško ali šestnajstiško… Kdor potrebuje več kot to: dokumentacija.

1.2 Kdaj uporabiti kaj?

Vsaka oblika ima še vedno svoje prednosti.

- V starejših oblikah (`%` in `format`) lahko shranimo vzorec v niz, npr. `vzorec = "Danes je %(vreme)s."` oz. `vzorec = "Danes je {vreme}."`. V novi tega ne moremo storiti, ker se niz `f"Danes je {vreme}"` izračuna (interpolira) takoj, na mesto. To je uporabno predvsem, kadar bi radi vzorec podali neki funkciji, ki bo vanj potem vstavila podatke in vse skupaj izpisala. Ni, da bi to počeli pogosto, lahko pa se zgodi. Včasih je tudi praktično shraniti vzorec v datoteko, ki jo v programu preberemo in vstavimo podatke. Z novo obliko to ne gre.
- V prvi in zadnji obliki (`%` in interpolacija) Python ne kliče funkcije, pri drugi, `format`, pa kličemo metodo. Nekoč sem naletel na situacijo - ki pa se je za noben denar ne morem spomniti (sploh mi ga pa niti nihče ne ponuja) - ko mi Python ni pustil uporabiti `format`, ker bi to zahtevalo klic, v onem kontekstu pa klic ni bil dovoljen. Mislim, da je bilo povezano z logiranjem oz. z napakami in Python tam ni hotel spreminjati sklada. Po spominu.

V praksi: predlagam, naj vsa nova koda uporablja interpolacijo. Navadite se jih tudi zato, ker boste tudi v drugih sodobnih jezikih uporabljali interpolacijo, le sintaksa bo v vsakem jeziku malo drugačna.

2 Kontrolni znaki v nizih

Kontrolne znake zapisujemo tako, kot ste vajeni od drugod:

- `\n`: nova vrstica,
- `\t`: tabulator,
- `\x41`: znaki, podani z ASCII kodo v šestnajstiškem zapisu (tole je veliki A),
- `\u0041`: znaki, podani z Unicode v šestnajstiškem zapisu (tudi tole je veliki A),
- `\N{GREEK SMALL LETTER PI}`: ime znaka v Unicode (tole očitno ni veliki A),
- še vse živo drugo,
- `\\`: backslash, AKA vzvratna poševnica.

Če imamo v nizu veliko vzvratnih poševnic in se nam jih ne podvaja, uporabimo r-nize (`r` kot *raw*). Pred narekovaj damo znak `r` in vzvratna poševnica bo le še vzvratna poševnica. Kot naj bi (čeprav najbrž ni) rekel Freud: *cigara je včasih vseeno samo cigara*.

Vzvratno poševnico često pišemo v imenih direktorijev. Pišite navadno; tudi Windowsi jo že dolgo razumejo. Pač pa se boste z njimi pretepali v regularnih izrazih. Da, tam pa potrebujemo r-nize.

Komentar si zaslužijo še kode za Unicode: tudi teh ne potrebujemo pogosto. Programi v Pythonu so navadno shranjeni kot Unicode, praviloma UTF-8, in lahko brez težav vsebujejo znak , kot smo prepričljivo videli tudi v enem gornjih nizov. Zato jih bomo z `\N{GREEK SMALL LETTER PI}` najbrž opisali približno tolikokrat, kot bomo veliki A opisali z `\x41`.

3 Delo z datotekami

Datoteke odpremo s funkcijo `open(ime_datoteke)`. Če ne podamo drugih argumentov, jo bo Python odprl kot besedilno datoteko, za branje in s kodiranjem, ki je privzeto za sistem in je v spodobnih sistemih UTF-8, na Windowsih pa bogvekaj, recimo cp1250. `open(ime_datoteke)` je torej v normalnih razmerah isto kot `open(ime_datoteke, "rt", encoding="utf-8")`. Argument `encoding` podamo poimensko, ker je vmes še en nezanimiv argument in ker je poimensko podajanje (vsaj tu) tako ali tako lepše.

3.1 Branje besedilnih datotek

Datoteka, ki jo odpremo za branje, se vede kot generator, ki vrača vrstice. Običajno bomo datoteko torej brali z

```
for vrstica in open(ime):
```

Druge metode, ki vas lahko zanimajo, so:

- `read()` prebere celotno datoteko v niz. Temu naj ne sledi `split("\n")`, ker je nesmiselno in vas poleg tega utegne zafrkniti, ker se bo zadnja vrstica tipično končala z `\n`, zato bo `split` na konec dodal še en prazen niz. Če že, naj temu sledi `splitlines()`, vendar tudi to ni smiselno - zaradi naslednje metode.
- `readlines()` prebere vse vrstice datoteke (oz. vse preostale vrtice, če nismo na začetku datoteke) v seznam nizov. Nizi vsebujejo tudi znak za novo vrstico, `\n`.
- `readline()` prebere eno (=naslednjo) vrstico datoteke.

3.2 Pisanje besedilnih datotek

Če želimo pisati v besedilno datoteko, jo odpremo z `open(ime_datoteke, "w")`, ki pobriše morebitno obstoječo datoteko s tem imenom, ali `open(ime_datoteke, "a")` (`a` kot *append*), ki dodaja na konec obstoječe datoteke. Namesto `w` in `a` lahko pišemo tudi `wt` ali `at`, vendar ni potrebno, ker je `t` (besedilna datoteka) tako ali tako privzeti način.

Edina res uporabna metoda za pisanje v datoteke je

- `write(s)`

Obstaja tudi

- `writelines(s)`, ki v datoteko zapiše seznam vrstic `s`. Ker `writelines` ne dodaja znakov `\n` (tako kot jih `readlines` ni brisal), jih moramo dodati sami. Potemtakem pa je hitreje preprosto napisati `write("\n".join(s))`.

3.3 Zapiranje datotek

Datoteke imajo metodo `close`. V praksi jo redko kličemo,

- ker se datoteka zapre sama, takrat ko *garbage collection* pospravi ime, ki se nanaša nanjo, torej tipično na koncu funkcije ali programa
- ker takrat, ko hočemo to izvesti po vseh pravilih, datoteko odpremo z `with`, ki pa sam skrbi za zapiranje. Več o tem pa v ločenem predavanju o *context managerjih*.

3.4 Binarne datoteke

Binarne datoteke odpremo tako, da v niz, s katerim podamo način odpiranja, dodamo `b`. Torej `open(ime, "rb")` oziroma `open(ime, "wb")`. Ko beremo binarno datoteko - kar očitno počnemo z `read` in ne `readline` ali `readlines`, saj binarne datoteke nimajo vrstic - ne dobimo niza (`str`) temveč bajte (`bytes`). Torej čisto drug podatkovni tip. Prav tako z `write` v binarne datoteke zapisujemo bajte, ne nizov.

To pa je tema za ločeno poglavje, saj podatkovni tip `bytes` ni povezan le z datotekami.