05 Funkcije

January 28, 2024

1 Funkcije

Opazili ste - in malo vam gre tudi na živce, bi rekel - da stalno ponavljamo ene in iste zadeve. Recimo: stalno je potrebno iskati *ključ seznama, ki mu pripada največja vrednost*. V smislu, imamo slovar, katerega ključi so predmeti in vrednosti njihove cene, naloga pa hoče, da poiščemo najdražji predmet.

[1]:
```
cene = {
    'slika': 45,
    'pozlačen dežnik': 29,
    'Meldrumove vaze': 78,
    'skodelice': 83,
    'kip': 107,
    'čajnik': 15,
    'srebrn jedilni servis': 63,
    'perzijska preproga': 21}
```

In potem se, vsakič znova, ubijamo z

[2]:
```
max_k = max_v = None

for k, v in cene.items():
    if max_v is None or v > max_v:
        max_v = v
        max_k = k
```

[3]:
```
print(max_k)
```

```
kip
```

Opravičujem se za kratka imena spremenljivk, ampak vsega tega imam res že dosti.

Imam dobro novico in dve slabi.

Dobra novica: obstaja funkcija `argmax`, ki ji podamo slovar in vrne ključ, ki pripada največji vrednosti.

Prva slaba novica: dobra novica ne drži. Funkciji `argmax` ne podamo slovarja, temveč seznam parov. Funkcija primerja druge elemente parov, poišče največjega in vrne pripadajoči prvi element. Vendar ta slaba novica ni tako usodna: funkciji pač podamo `d.items()` (če je `d` naš slovar). Tako bo dobila `[('slika', 45), ('pozlačen dežnik', 29), ('Meldrumove vaze', 78),`

`('skodelice', 83), ('kip', 107), ('čajnik', 15), ('srebrn jedilni servis', 63), ('perzijska preproga', 21)]` in vrnila `'kip'`, saj je to prvi element, ki mu pripada največji drugi element.

Druga slaba novica: dobra novica sploh ne drži. Take funkcije ni. Fake novica.

Vedno rad citiram nepozabnega srednješolskega profesorja matematike Francija Oblaka: "*Česar ni, se pa nar'di.*" Naredimo torej takšno funkcijo.

Oglejmo si celico 2, v kateri smo računali ta naš slavni maksimum. V njej je zapisano vse, kar mora delati ta funkcija. Seveda pa si želimo, da bi poiskala maksimum v poljubnem slovarju, ne nujno v `cene`. Poleg tega se bomo dogovorili, da tisto, v čemer išče maksimum, ne bo slovar temveč seznam parov. Imeli bomo torej nekaj takšnega:

```
max_k = max_v = None

for k, v in s:
    if max_v is None or v > max_v:
        max_v = v
        max_k = k
```

To je le prepisana vsebina gornje celice, le `cene.items()` sem zamenjal s `s`. Ta `s` bo tisto, kar je funkcija dobila kot argument.

Kaj se to pravi *`s` bo tisto, kar je funkcija dobila kot argument*?! No, pač, če bomo poklicali `argmax(cene.items())`, bo funkcija `s` vseboval `cene.items()`.

Očitno bo potrebno postoriti (vsaj) še dvoje. Troje, pravzaprav. Četvero. (To že postaja podobno Španski inkviziciji)!

1. Te vrstice je treba nekako "zgrupirati", povedati Pythonu, da bo to koda neke funkcije.
2. Funkciji je potrebno dati ime (`argmax`).
3. Povedati je potrebno, da bo zahtevala en argument.
4. Povedati je potrebno, pod kakšnim imenom bomo znotraj funkcije videli (dostopali do…) podatke, danega kot argument.

Vse to storimo praktično v eni vrstici.

[4]:
```
def argmax(s):
    max_k = max_v = None

    for k, v in s:
        if max_v is None or v > max_v:
            max_v = v
            max_k = k
```

1. Z besedo `def` povemo, da sledi definicija ganz nove funkcije. (Dandanašnja mladina sploh ne zna več prave slovenščine, zato jo je treba kdaj spomniti, da obstajajo tudi druge, bolj slovenske besede kot *totally*.)
2. Slediti ime funkcije, torej `argmax`.
3. Nato pridejo oklepaji in v njih toliko imen, kolikor argumentov zahteva funkcija. Med imena

(če jih je več) postavimo vejice. Če funkcija ne mara argumentov, vseeno napišemo oklepaje, čeravno so prazni oklepaji videti neumno. Tu so možne še vsakršne druge komplikacije, kot recimo argumenti, ki jih lahko izpustimo in imajo privzeto vrednost, ali pa funkcije, ki prejemajo poljubno število argumentov, funkcije, ki zahtevajo, da argumente poimenujemo. Začetni tečaj programiranja, sploh za študente, ki najbrž ne bodo profesionalni računalnikarji, vsega tega ne potrebuje.

4. To, kar funkcija dobi kot argument(e), vidimo pod imeni, ki so navedena v oklepajih. Naša funkcija je navedla eno ime, `s`, torej kot argument pričakuje eno stvar in znotraj funkcije se bomo na to stvar sklicevali z imenom `s`. Pod kakšnim imenom je ta stvar znana zunaj, nas niti ne zanima niti (če nismo res vztrajni in dobro poznamo Python) tega ne moremo izvedeti. In prav je tako.

Preskusimo, če deluje!

[5]:
```
print(argmax(cene.items()))
```

```
None
```

`None`? Kakšen `None`?! Mar nismo obljubili, da bo funkcija vračala … Aja. Kje pa piše, kaj ta funkcija vrne? V funkciji živi kup spremenljivk - ne le `max_v` in `max_k` temveč tudi `k` in `v`. Python ne more vedeti, katera od njih naj bo rezultat funkcije. Če sploh katera; obstajajo tudi funkcije (recimo `print`), ki ne vračajo ničesar. Da, Python si je očitno predstavljal, da ta funkcija ne bo vračala ničesar, pa se nam je zato posmehnil s tem `None`.

V seznam potreb španske inkvizicije torej dodamo še

5. Povedati moramo, kaj funkcija vrne.

in dopolniti funkcijo.

[6]:
```
def argmax(s):
    max_k = max_v = None

    for k, v in s:
        if max_v is None or v > max_v:
            max_v = v
            max_k = k
    return max_k
```

In jo preskusiti.

[7]:
```
print(argmax(cene.items()))
```

```
kip
```

To je to: z besedo `return` povemo, kaj naj funkcija vrne.

1.1 Več `return`-ov

Napišimo funkcijo `dragoceni(s)`, ki vrne predmete, katerih cene so večje od 70. Točneje, funkcija prejme seznam parov, pri čemer predpostavljamo, da je drugi element para število. Vrne tiste prve

elemente, pri katerih je drugi element večji od `70`. Rezultat morajo biti Meldrumove vaze, skodelice in kip.

[14]:
```
def dragoceni(s):
    for k, v in s:
        if v > 70:
            return k
```

[15]:
```
predmeti = dragoceni(cene.items())

print(predmeti)
```

```
Meldrumove vaze
```

Kaj pa skodelice in kip?

Nekatere je najbrž zaskrbelo že prej, nekateri razumejo zdaj, nekateri pa še bodo. Naloga ni bila ravno najboljše definirana. *Kaj* točno naj bi funkcija vračala? Niz? Več nizov? Se pravi, kakšnega tipa naj bi bili `predmeti`?

Besedilo naloge je nekako impliciralo, da bo šlo za več nizov, ni pa jasno povedala, kako naj bi bilo teh več nizov shranjenih. Naivno smo postavili `return` v zanko in pričakovali, da se bo izvedel večkrat … Python pa bo že nekaj naredil s temi več rezultati. No, ne bo.

Trije nauki.

1. Ko se Python naleti na `return`, se izvajanje funkcije konča. Funkcija vrne, kar vrne in to je to. `return` je tako prekinil zanko.
2. Tako kot smo ob sestavljanju slovarja sklenili, da bomo poprej razmislili, kaj bodo njegovi ključi in njegove vrednosti, moramo tudi ob sestavljanju funkcije pomisliti, kaj (na primer: kakšnega tipa) bodo njegovi argumenti in njegov rezultat. Tu nismo.
3. Funkcija lahko vrača tudi druge stvari, recimo sezname.

Tretji nauk še ni naučen, zato le hitro popravimo funkcijo.

[12]:
```
def dragoceni(s):
    dragi = []
    for k, v in s:
        if v > 70:
            dragi.append(k)
    return dragi
```

[13]:
```
predmeti = dragoceni(cene.items())

print(predmeti)
```

```
['Meldrumove vaze', 'skodelice', 'kip']
```

Napišimo še eno podobno funkcijo s podobnim naukom: funkcijo `obstajajo_dragoceni`, ki vrne `True`, če obstaja kak dragocen predmet in `False`, če ne.

[16]:
```
def obstajajo_dragoceni(s):
    dragi = []
    for k, v in s:
        if v > 70:
            return True
    return False
```

Razumemo? Ta funkcija ima dva `return`-a. Kot smo uvideli, se lahko zgodi le eden - pač tisti, na katerega naletimo prej. Za vsak predmet preverimo, ali je dragocen. Če je, vrnemo `True`. Kdaj pa bomo vrnili `False`? Pač, če nikoli nismo vrnili `True`. Če torej ni bilo nobenega dragocenega predmeta.

1.2 Funkcije se kličejo med sabo

V programih, ki smo jih pisali doslej, smo klicali različne Pythonove funkcije. Tudi v svojih funkcijah jih lahko. V gornjih primerih tega sicer ni bilo veliko, klicali smo le `append`, a najbrž ne bi nihče dvignil obrvi, če bi poklicali tudi kakšen `len` ali `print` ali karkoli drugega. Kaj pa naše funkcije? Lahko v svojih funkcijah kličemo svoje funkcije. Najsevedajše da. Večje programe gradimo kot skladovnice lastnih funkcij; preprostejše funkcije kombiniramo v vedno zapletenejših, nivoje in nivoje visoko. Pri tem predmetu česa prav zapletenega ne bo, preprost primer pa lahko vidimo že tu: funkcija `dragoceni` je splošnejša od funkcije `obstajajo_dragoceni`. Prva vrne seznam dragocenih predmetov, druga pa pove le, ali dragoceni predmeti obstajajo. Druga funkcija bi lahko preprosto poklicala prvo in preverila, ali le-ta vrne prazen seznam.

[17]:
```
def obstajajo_dragoceni(s):
    # Ne programirajte tako!!!
    if dragoceni(s) != []:
        return True
    else:
        return False
```

Današnji primeri so res bogati. Vsak kaže par stvari. Tale vsaj dve. Uh, tri. Inkvizicija.

1. Programiranje si boste olajšali, če boste razmišljali o tem, kaj že imate in to uporabljali. Še več; včasih boste napisali dve podobni funkciji in če boste pametni, boste tisto, kar jima je skupno, izločili v tretjo funkcijo. Tako boste lahko onidve funkciji skrajšali tako, da boste poklicali, kar je skupnega.
2. Svarilo: funkcija `obstajajo_dragoceni` je s tem postala počasnejša! Prej je zanka `for` tekla do prvega dragocenega predmeta in se tam končala, prekinil jo je `return`. Zdaj `obstajajo_dragoceni` sicer ne vsebuje nobene zanke, vendar sproži zanko v `dragoceni`, ta pa vedno teče do konca, pa še predmete zlaga v nek seznam, ki ga potem zavržemo, saj nas zanima le, ali je prazen ali ne. Nas to boli? Odvisno od situacije. Če takole, za lastne potrebe analiziramo neke kratke datoteke, ni problema. Če bi bili podatki ogromni in če bi funkcijo, kot je `obstajajo_dragoceni` klicali velikokrat, pa bi jo raje pustili takšno, kot je bila.
3. Tako se ne programira!

Nadaljujmo s tretjo točko. `dragoceni(s) != []` je logični izraz, katerega rezultat je **že** `True` ali `False`. Funkcija bi se pravilneje (ne v smislu tega, da vrača pravilen rezultat, temveč v smislu, uh,

smiselnosti) glasila

[18]:
```
def obstajajo_dragoceni(s):
    return dragoceni(s) != []
```

Nauk 4: funkcije so lahko tudi zelo kratke. V življenju sem napisal že veliko funkcij, dolgih eno samo vrstico. Le da je ta vrstica pogosto daljša in bolj zapletena od te tu.

1.3 Več argumentov

Gornje funkcije ne upoštevajo inflacije. Danes je 80 veliko, jutri pa ne boste mogli v Hofer ali Lidl, ne da bi tam pustili vsaj toliko. (Za arhiv: tole pišem 7. novembra 2023. Bomo videli, kako se bo staralo.)

Napišimo boljši funkciji `dragoceni` in `obstajajo_dragoceni`: poleg seznama naj prejmeta še mejo "dragocenosti". Če bomo poklicali `dragoceni(cene.items(), 70)` bomo dobili enak rezultat kot zdaj, če `dragoceni(cene.items(), 100)` pa samo še kip.

[19]:
```
def dragoceni(s, meja):
    dragi = []
    for k, v in s:
        if v > meja:
            dragi.append(k)
    return dragi


def obstajajo_dragoceni(s, meja):
    return dragoceni(s, meja) != []
```

1.4 Argumenti s privzetimi vrednostmi

Mimogrede se naučimo še to: pogosto pišemo funkcije, ki "skoraj ne potrebujejo" nekega argumenta. Recimo, da bi funkcijo, kot je `dragoceni`, klicali na sto mestih v programu in meja bi bila praktično vedno `70`, le v parih primerih pa kaj drugega. V tem primeru bi argumentu `meja` nastavili prizveto vrednost na `70`. Argument bi s tem postal *opcijski* - lahko ga podamo ali pa tudi ne. Če ga izpustimo, bi imel vrednost 70.

To je preprosto narediti.

[21]:
```
def dragoceni(s, meja=70):
    dragi = []
    for k, v in s:
        if v > meja:
            dragi.append(k)
    return dragi


def obstajajo_dragoceni(s, meja=70):
    return dragoceni(s, meja) != []
```

[23]:
```
dragoceni(cene.items(), 100)
```

[23]:
```
['kip']
```

[24]:
```
dragoceni(cene.items())
```

[24]:
```
['Meldrumove vaze', 'skodelice', 'kip']
```

1.5 Več rezultatov ... in nič rezultatov

Kaj se zgodi, če funkcija nima ukaza `return` (ali pa ga ima, vendar med izvajanjem funkcije nikoli ni prišlo do njega, ker je bil, recimo, skrit znotraj nekega `if`, čigar pogoj nikoli ni bil resničen), že vemo. Funkcija v tem primeru vrne `None`.

Funkcije vedno vrnejo natančno eno stvar. Ne nič, ne dve, ne pet. Eno.

To je slabo. Ampak ne preveč, saj se lahko na to pravilo preprosto požvižgamo. Kot prvo: če ne vrnemo ničesar, funkcija vrne `None`. S tem sicer vrne (natančno) eno stvar, vendar nas to nič ne boli. Naj jo.

Enako se požvižgamo na pravilo, da sme funkcija vrniti le eno stvar in jih mirno vrnemo več.

[28]:
```
def argminmax(s):
    min_k = min_v = None
    max_k = max_v = None

    for k, v in s:
        if min_v == None or v < min_v:
            min_v = v
            min_k = k
        if max_v == None or v > max_v:
            max_v = v
            max_k = k
    return min_k, max_k
```

[29]:
```
najm, najv = argminmax(cene.items())

print(najm)
print(najv)
```

```
čajnik
kip
```

Razumemo, kaj se dogaja?

`min_k, max_k` je v resnici terka.

[32]:
```
argminmax(cene.items())
```

[32]:
```
('čajnik', 'kip')
```

Ko smo spoznali terke, smo omenili, da jih smemo pisati brez oklepajev, vendar to počnemo v redkih primerih. Tu je primer, ko terke *vedno* pišemo brez oklepajev. (To je spet nepisano pravilo: lahko bi dodali tudi oklepaja). Delamo se (predstavljamo si), da je `return` vrnil dve stvari (čeprav je v resnici eno, namreč terko z dvema elementoma). Ob klicu funkcije rezultat takoj razpakiramo.

Pa imamo: funkcije, ki navidez vračajo več vrednosti. Ta konkretno, dve.

Zato se lahko na pravilo, da funkcije vedno vračajo natančno eno stvar, preprosto požvižgamo. Navidez vračajo tudi nobene ali več. Da je v resnici le ena, nihče ne bo opazil.