

## Obvezna naloga

### podatek(kljuc, podatki)

Napiši funkcijo `podatek(kljuc, podatki)`, ki dobi neko vrednost `kljuc` in nek seznam parov. Med njimi mora poiskati tisti, par, pri katerem je prva vrednost enaka podanemu ključu; funkcija mora vrniti drugo vrednost.

- Klic `podatek("Berta", [("Ana", 156), ("Berta", 167), ("Cilka", 160)])` vrne 167.
- Klic `podatek(42, [(42, "6 * 8"), (33, "11 * 3")])` vrne `"6 * 8"`.
- Klic `podatek("Koper", [('Piran', (0, 0)), ('Koper', (8, 2)), ('Žiri', (39, 60))])` vrne `"Koper"`.
- Klic `podatek("Žiri", [("Piran", 23), ("Koper", 4), ("Žiri", 481)])` vrne 481.

Če ključa ne najde, (npr. v klicu `podatek("Žiri", [("Piran", 23), ("Koper", 4)])`) mora vrniti `None`.

**Rešitev** Gremo čez podatke. Spremenljivki iz para bomo imenovali `k` in `v`. V opravičilo, zakaj to ni neskladno z mojim stalnim teženjem o tem, da je potrebno spremenljivki pametno poimenovati, naj povem, da je ta par kar običajen: `k` in `v` se v Pythonu stalno uporablja za kombinacijo *key* in *value*, kar slučajno lepo sovпада s slovenskim *ključ* in *vrednost*.

Če naletimo na ključ, ki se ujema s podanim, vrnemo vrednost. Če se zanka izteče neuspešno vrnemo `None`.

```
def podatek(kljuc, podatki):
    for k, v in podatki:
        if k == kljuc:
            return v
    return None
```

Eksplisitni `return None` na koncu bi lahko tudi izpustili, saj funkcije, ki ne vrnejo ničesar, vrnejo `None`. Vendar se šteje za lepo prakso, da v primeru, da funkcija *včasih* vrne vrednost, le-ta vrne vrednost `None` tudi, kadar nima česa vrniti. Da se ve, da nismo pozabili.

### razdalja(t1, t2)

Napiši funkcijo `razdalja(t1, t2)`, ki prejme dve terki, ki predstavljata koordinati točk na ravnini. Vrniti mora evklidsko razdaljo med točkama. Klic `razdalja((0, 0), (3, 4))` mora vrniti 5.

**Rešitev** Razpakiramo in izračunamo

```
from math import sqrt # najboljše je, da damo to na začetek programa
```

```
def razdalja(t1, t2):
    x1, y1 = t1
```

```

x2, y2 = t2
return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

```

**oddaljenost(kraj1, kraj2, koordinate)**

Napiši funkcijo `oddaljenost(kraj1, kraj2, koordinate)`, ki prejme imeni dveh krajev in seznam, ki vsebuje koordinate krajev v takšni obliki, kot ga vidiš v tretjem primeru pri prvi funkciji. Funkcija mora vrniti evklidsko razdaljo med krajema. (Dober nasvet: funkcija naj kliče prejšnji funkciji.) Predpostaviti smeš, da se koordinate obeh krajev nahajajo v seznamu.

**Rešitev** Če uporabimo funkciji `podatek` in `razdalja`, opravimo s tem kar v enem koraku.

```

def oddaljenost(kraj1, kraj2, koordinate):
    return razdalja(podatek(kraj1, koordinate), podatek(kraj2, koordinate))

```

**visina(kraj1, kraj2, visine)**

Napiši funkcijo `visina(kraj1, kraj2, visine)`, ki prejme imeni dveh krajev in višine krajev v seznamu, kot ga vidiš v četrtem primeru pri prvi funkciji. Funkcija naj vrne višinsko razliko (vzpon) od prvega kraja do drugega; če je prvi višji, funkcija vrne negativno vrednost. Spet smeš predpostaviti, da sta višini obeh krajev znani.

**Rešitev** Še lažje.

```

def visina(kraj1, kraj2, visine):
    return podatek(kraj2, visine) - podatek(kraj1, visine)

```

Na podlagi izkušenj iz prejšnjih let sem domneval - in nisem se motil - da bo veliko študentov kompliciralo takole.

```

def visina(kraj1, kraj2, visine):
    razlika = podatek(kraj2, visine) - podatek(kraj1, visine)
    if razlika > 0:
        return razlika
    else:
        return -razlika

```

To je seveda narobe: če je `razlika` negativna in vrnemo `-razlika`, vrnemo pozitivno razliko, kadar je le-ta negativna. Ko odkrijejo to napako, jo popravijo z

```

def visina(kraj1, kraj2, visine):
    razlika = podatek(kraj2, visine) - podatek(kraj1, visine)
    if razlika > 0:
        return razlika

```

```

else:
    return -abs(razlika)

```

To deluje, ni pa zelo smiselno. Navodilo pravi samo, da razliko vedno merimo od prvega do drugega kraja in včasih je ta pač negativna. Ampak rezultat takega navodila me vedno zabava: študenti naredijo napako prav zato, ker poskušajo upoštevati tisto, kar bi brez mojega opozorila spregledali (in bi vse delalo). :)

**cas(s, h)**

Napiši funkcijo **cas(s, h)**, ki vrne čas, ki ga porabi nek kolesar, da prevozi pot dolžine **s** (v kilometrih), ki se dvigne za **h** (v metrih; **h** je lahko tudi negativen, če gre za spust). Čas izračunamo tako:

- kolesar v osnovi porabi 2,4 minute za kilometer razdalje.
- Če gre za dvig, se čas poveča še za 1 / 12 minute za vsak meter višine;
- če gre za spust, pa se čas zmanjša za 1 / 250 minute za vsak meter spusta.

Primeri:

- Klic `cas(2, 300)` vrne 29,8: za 2 km porabi  $2 * 2,4 = 4,8$  minut, poleg tega pa porabi  $1 / 12 * 300 = 25$  minut, skupaj 29,8 minut.
- Klic `cas(10, -100)` vrne 23,6: za 10 km porabi  $10 * 2,4 = 24$  minut, vendar se to zmanjša za  $1 / 250 * 100 = 0,4$  minute, skupaj 23,6 minut.

(Posebej drugi primer kaže, da vsaka podobnost te formule z resničnimi dogodki ni zgolj naključna temveč tudi ne prav zelo verjetna.)

**Rešitev** No, tole pa zahteva nek pogoj in ločeno obravnavo pozitivnih in negativnih višin. Naredimo lahko, recimo, tako:

```

def cas(s, h):
    if h > 0:
        return 2.4 * s + 1 / 12 * h
    else:
        return 2.4 * s + 1 / 250 * h

```

**cas\_med(kraj1, kraj2, koordinate, visine)**

Napiši funkcijo **cas\_med(kraj1, kraj2, koordinate, visine)**, ki vrne čas, ki ga bo kolesar potreboval za vožnjo od prvega do drugega kraja. Pomen argumentov in formule so takšne kot v prejšnjih nalogah. (Uporabljal funkcije, ki jih že imaš!)

**Rešitev** Kot na opominja navodilo, moramo le poklicati funkcije, ki jih že imamo.

```

def cas_med(kraj1, kraj2, koordinate, visine):
    razdalja = oddaljenost(kraj1, kraj2, koordinate)
    razlika = visina(kraj1, kraj2, visine)
    return cas(razdalja, razlika)

```

Šlo bi tudi tako:

```
def cas_med(kraj1, kraj2, koordinate, visine):  
    return cas(oddaljenost(kraj1, kraj2, koordinate),  
               visina(kraj1, kraj2, visine))
```

Vendar to ni bistveno krajše, praktično nič hitrejše, je pa manj berljivo. Prva različica je boljša, ker poimenuje vmesne vrednosti in bolj jasno pove, kaj podajamo funkciji `cas`.

```
skupni_cas(kraji, koordinate, visine)
```

Napiši funkcijo `skupni_cas(kraji, koordinate, visine)`, ki prejme seznam krajev, skozi katere bo šel kolesar, in vrne skupni čas, ki ga bo porabil za to pot.

**Rešitev** Iti moramo prek vseh parov zaporednih krajev. To lahko dosežemo z motoviljenjem z indeksi, za Python zglednejša pa je tale rešitev:

```
def skupni_cas(kraji, koordinate, visine):  
    skupni = 0  
    for kraj1, kraj2 in zip(kraji, kraji[1:]):  
        skupni += cas_med(kraj1, kraj2, koordinate, visine)  
    return skupni
```

Čez ne tako dolgo časa pa bomo vedeli, da se to bolj zares sprogramira tako:

```
def skupni_cas(kraji, koordinate, visine):  
    return sum(cas_med(kraj1, kraj2, koordinate, visine)  
               for kraj1, kraj2 in zip(kraji, kraji[1:]))
```

## Dodatna naloga

Kolesar bi rad čimbolj skrajšal čas potovanja. Izpustiti sme en kraj - lahko prvega, lahko zadnjega, lahko kateregakoli vmes.

Napiši funkcijo `skrajsaj(kraji, koordinate, visine)`, ki vrne seznam brez tega kraja. Klic `skrajsaj(["Piran", "Ljubljana", "Koper"], koordinate, visine)` vrne `["Piran", "Koper"]`. Enak rezultat vrneta tudi klica `skrajsaj(["Ljubljana", "Piran", "Koper"], koordinate, visine)` in `skrajsaj(["Piran", "Koper", "Ljubljana"], koordinate, visine)`.

### Rešitev

Dodatna naloga ni bila posebej težka. Edina sitnost je bila sestavljanje skrajšanega seznama.

Nekateri so šli z zanko čez kraje in z `remove` odstranjevali posamezne kraje iz seznama. Ker bi vas rad naučil o tem, kako nevarna funkcija je `remove`, sem v testih namerno sestavil tudi pot, v katerem se en kraj ponovi. `remove` odstrani prvo ponovitev.

V podobno past so se ujeli tisti, ki so sestavljali nov seznam tako, da so prepisali vanj vse elemente razen tega, ki ga hočejo izločiti. Ti so odstranili obe ponovitvi kraja.

Vsekakor je bilo tu potrebno delati z indeksi. Jaz sem se lotil tako, da sem s pomočjo rezin sestavljal nove sezname, v katerih je manjkal prvi, drugi, tretji element in tako naprej.

```
def skrajsaj(kraji, koordinate, visine):
    naj_cas = skupni_cas(kraji, koordinate, visine)
    naj_pot = kraji
    for i in range(len(kraji)):
        predlog = kraji[:i] + kraji[i + 1:]
        cas = skupni_cas(predlog, koordinate, visine)
        if cas < naj_cas:
            naj_cas, naj_pot = cas, predlog
    return naj_pot
```

Za začetek predpostavimo, da je najboljša rešitev kar celotna pot. Tekom zanke se bo izkazalo, da to ni res. Če je pot slučajno prazna, pa bo funkcija vrnila to, prazno pot, kar je tudi dobro.

Funkcija je, v primerjavi z gornjimi kar dolga. Čez neveliko časa bomo znali napisati krajšo.

```
def skrajsaj(kraji, koordinate, visine):
    return min((kraji[:i] + kraji[i + 1:] for i in range(len(kraji))),
               key=lambda x: skupni_cas(x, koordinate, visine))
```

Obe rešitvi počneta isto, le da v drugi rešitvi funkcija `min` počne tisto, kar smo morali v prvi rešitvi sprogramirati sami.

Obe rešitvi sta tudi nekoliko počasni, ker vedno znova računata trajanje (skoraj) celotne poti. Hitrejša, a daljša rešitev je, da opazujemo zaporedne trojke krajev in preverjamo, pri kateri trojki prihranimo največ, če izpustimo srednji kraj. Pri tem je sitno predvsem, da moramo ločeno obravnavati začetek in konec.

```
def skrajsaj(kraji, koordinate, visine):
    naj_prihranek = cas_med(kraji[0], kraji[1], koordinate, visine)
    naj_indeks = 0
    prihranek = cas_med(kraji[-2], kraji[-1], koordinate, visine)
    if prihranek > naj_prihranek:
        naj_prihranek = prihranek
        naj_indeks = len(kraji) - 1
    for i, (k1, k2, k3) in enumerate(zip(kraji, kraji[1:], kraji[2:])):
        prihranek = (cas_med(k1, k2, koordinate, visine)
                     + cas_med(k2, k3, koordinate, visine)
                     ) - cas_med(k1, k3, koordinate, visine)
        if prihranek > naj_prihranek:
            naj_prihranek = prihranek
```

```
naj_indeks = i + 1
return kraji[:naj_indeks] + kraji[naj_indeks + 1:]
```

## Dodatni izziv

Recimo, da bi v vseh krajih v seznamu `koordinate` zabili visok jambor in nato okrog Slovenije napeli zelo dolgo elastiko. Ta bi se dotikala le nekaterih jamborjev - tistih, "na robu". Ljubljane gotovo ne, saj je preveč v sredini, gotovo pa bi se dotikala Murske Sobote, ki štrli ven.

Napiši funkcijo, ki prejme seznam koordinat in vrne seznam krajev (po vrsti, začenši od poljubnega kraja), ki bi se jih elastika dotikala.

## Rešitev

Rešitev te naloge pa posebej, ob predavanjih za tiste, ki znajo že malo več.

---

This task includes tests. Your solution will be accepted as correct only if it passes them. See instructions for running tests.

## Mandatory Task

1. Implement a function `podatek(kljuc, podatki)`, which receives a certain value `key` and a list of pairs (`podatki`). The function must find a pair in which the first value equals the key, and it must return the second value of the pair.

- Call `podatek("Berta", [("Ana", 156), ("Berta", 167), ("Cilka", 160)])` returns 167.
- Call `podatek(42, [(42, "6 * 8"), (33, "11 * 3")])` returns "6 \* 8".
- Call `podatek("Koper", [('Piran', (0, 0)), ('Koper', (8, 2)), ('Žiri', (39, 60))])` returns (8, 2).
- Call `podatek("Žiri", [("Piran", 23), ("Koper", 4), ("Žiri", 481)])` return 481.

If the key is not found (as in `podatek("Žiri", [("Piran", 23), ("Koper", 4)])`) the function must return `None`.

1. Implement a function `razdalja(t1, t2)`, which gets two tuples that represent coordinates of two points in a plain. It must return the Euclidean distance between them. The call `razdalja((0, 0), (3, 4))` must return 5.
2. Implement a function `oddaljenost(kraj1, kraj2, koordinate)`, which gets names of two places and a list with names and coordinates in the same form as in the third example for the first function. The function must

return the Euclidean distance between the two places. (A good advice: the function should call the first two functions.)

You may assume that the list includes both places.

3. Implement a function `visina(kraj1, kraj2, visine)`, which gets names of two places and list of heights above sea level as in the fourth example for the first function. The function must return the ascent from the first place to the second; the ascent is of course negative if the second place is lower than the first. You may again assume that both places appear in the list.
4. Implement a function `cas(s, h)`, which returns time needed by a cyclist for a path of length `s` kilometers with `h` meters of ascent (`h` can also be negative). The time is computed using the following formulae:
  - the cyclist needs, basically, 2.4 minutes per kilometer.
  - In case of ascent, the time is increased by  $1 / 12$  minutes per meter of ascent;
  - in case of descent, the basic time is reduced by  $1 / 250$  minute per meter of descent.

Examples:

- Call `cas(2, 300)` returns 29.8: for 2 km she needs  $2 * 2.4 = 4.8$  minutes, which is increased by  $1 / 12 * 300 = 25$  minutes because of 300 m ascent.
- Call `cas(10, -100)` returns 23.6: for 10 km she needs  $10 * 2.4 = 24$  minutes, which is decreased by  $100 / 250 = 0.4$  minutes because of descent.

(The second example clearly shows that any similarity of this formulae with real times is purely coincidental and not very probable.)

1. Implement a function `cas_med(kraj1, kraj2, koordinate, visine)`, that return time that the cyclist will need from the first to the second place. The meaning of arguments and the formulae are the same as before. (And so is the advice: use the functions that you already have.)
2. Implement a function `skupni_cas(kraji, koordinate, visine)`, which gets a list of places that the cyclist will traverse, and returns the total time needed for the trip.

## Extra tasks

The cyclist would like to decrease the time needed for the trip. He may skip one single place -- the first, the last or any in between.

Implement a function `skrajsaj(kraji, koordinate, visine)`, which returns a list without that place. The order of other places must stay the same. Call `skrajsaj(["Piran", "Ljubljana", "Koper"], koordinate, visine)`

returns `["Piran", "Koper"]` (Piran and Koper are nearby towns on the seaside, and Ljubljana is 100 km away). The same result is returned by `skrajsaj(["Ljubljana", "Piran", "Koper"], koordinate, visine)` and `skrajsaj(["Piran", "Koper", "Ljubljana"], koordinate, visine)`.

### Extra challenge

(Solve this for your amusement; it does not affect the grade.)

Assume a large mast is set at each place in the list `koordinate`. Then we stretch an elastic rope around Slovenia. This rope surely does not touch Ljubljana, which is a rather central town, but it touches Murska Sobota, which is in the corner, not far from Austria and Hungary.

Write a function that gets a list of coordinates and returns a list of towns (in order, but starting from any) that the rope will touch.