Najprej si poglejmo učilnico. Za vse začetnike lahko tukaj najdete čudovito knjigo Python za programiranje. Poda odlično iztočnico za učenje. Priporočam vam, da si v začetnih tednih vzamete par dni in se res dobro naučite Python. V nasprotnem primeru boste imeli skozi celoten semester velike težave z delom na domačih nalogah. Tukaj je tudi povezava na PyCharm, ki je odličen IDE.

Za vse tiste, ki uporabljate Linux ali macOS imate Python že inštaliran. Opozorit vas morem le na to, da obstajata dve verziji Pythona: 2+ in 3+. Prva je starejša in ni kompatibilna z novejšo. Zaradi velike razširjenosti pa je uradno podprta še vse do leta 2020. Mi bomo pri tem predmetu uporabljali novejšo verzijo. Načeloma veljajo vse vezje od 3.3 naprej za stabilne in se niso preveč spreminjale, da pa ne bi prišlo do nepotrebnih napak pri testiranju je boljše, da se poenotimo in rečemo, da razvijamo v verziji 3.6 ali 3.7. Uporabniki Windowsa namestite Python s pomočjo inštalacijskega programa, ki ga najdete na spletu.

Python je visoko-nivojski programski jezik, ki se interpretira. Omogoča pisanje tako objektno kot funkcijsko orientirane kode. Znan je predvsem po berljivosti kode. Je dinamično tipiziran, kar pomeni, da spremenljivke nimajo določenih primitivnih tipov kot so int, float, char... Vse v Pythonu je objekt.

Če odpremo Python v lupini dobimo interaktivno Python lupini, kjer lahko izvajamo kodo vrstico po vrstici. Interpreter pa nam vrača rezultate. Začnimo najprej z nekaj enostavnimi primeri.

```
1+1
2**10
3+4j
print(’abc’)
a = 10, a + 20
[’abc’, 24, None, True]
```

Premaknimo se v PyCharm. Najprej ustvarimo novi projekt. Ker smo prej ugotovili, da imamo lahko več Python interpreterjev moramo nastavit katerega bomo uporabljali. To naredimo v nastavitvah projekta. Za nas Linux in masOS uporabnike je pomembno, da ustvarimo novo virtualno okolje. Kaj točno to je bom razložil kasneje. Na levi strani imamo seznam datotek, na desni se bo prikazovala koda. Preden začnemo pisat kodo v datoteke pa si poglejmo še spodnjo orodno vrstico. Tukaj namreč najdemo Python interaktivno lupino. Ta pa nam omogočen še malo več kot tista od prej. Imamo namreč ’autocomplet’ ter seznam trenutno nastavljenih spremenljivk ter njihove vrednosti.

Preden začnemo pisat programe v datoteko si poglejmo še nekaj primerov kode. Za začetek si poglejmo sezname, tej boste namreč veliko uporabil pri domačih nalogah:

```
s = [3, 1, 2, 4]
len(s)
s.append(5)
s.remove(3)
s.sort()
s[0]
s[-2]
s[1:3]
s[:-1]
```

Zelo podobno kot seznam se obnaša tudi podatkovni tip terka le, da je njegova lastnost nespremenljivost (immutable). Ne moremo dodajat ali odstranjevat elementov. Podobna struktura so

tudi množice. Pri njih imamo lahko vedno samo eno primer elementa. Podobno kot seznam se obnaša tudi nizi, ki je če pomislimo samo seznam črk. Veliko bomo pa tudi uporabljali podatkovno strukturo slovar.

```
starost = {}
starost['Robi'] = 24
starost['Jaka'] = 23
print(starost['jaka'])
starost.keys()
starost.values()
starost.items()
```

Nadaljujemo programiranje v datoteki. Ustvarimo novo datoteko. Tradicionalno jo poimenujemo foo, našo prvo funkcijo pa bar.

```
def bar(x):
    return x**2

if __name__ == '__main__':
    print(bar(5))
```

Definirali smo funkcijo, ki prejme en argument ter vrne njen kvadrat. Telo funkcije je zamaknjeno s tabulatorjem ali štetimi presledki. PyCharm avtomatsko tabulatorje preslika v presledke. Ta del vam je zagotovo očiten. Drugi del je if stavek, ki pa mogoče ni čist jasen na prvi pogled. V spremenljivki __name__ je shranjeno ime programa. Ta je enak '__main__', kadar poganjamo skripto kot glavni program. Spremenljivka je nastavljen tudi na ime skripte, kadar se funkcije in razredi uvažajo v drugo kodo. Več o tem tudi še kasneje. Mimogrede, Python zna avtomatsko testirat ali je podan izraz resničen (True). Tako na primer lahko namesto a==True napišeš samo a.

Sedaj, ko ste dobit malo občutka koko delat s Pythonmo nadaljujmo na začetek reševanja domače naloge. Poglejmo si navodila za prvo domačo nalogo in kaj pravijo zapiski. Ustvarimo novo datoteko naloga1.py kamor bom pisal kodo. Pogledamo si v zapiske, kako se izvaja hierarhično razvrščanje v skupine. Pretipkam v Python komentarje, katere funkcionalnosti moremo sprogramirat.

Pričnem z branjem datoteke. Pokažem jim kako zgleda datoteka. Vidimo, da so stolpci ločeni z tabulatorjem. Pričnem najprej v konzoli. Kar je dobro v tej konzoli je tudi to, da lahko uporabljamo relativen dostop do datotek. Odprem datoteko. Python ima bogat nabor vgrajenih funkcij ('python builtin functions' in 'python reading files'). Prav tako bogat je tudi nabor metod objekta niz ('python string object').

```
f = open('1.domaca/data/grades.csv')
s = f.readline()
f.close()
s = s.rstrip()
s = s.split('\t')
header = s[1:]
for line in f:
    print(line)
```

Prestavim v datoteko:

```
with open(file_name) as f:
    header = f.readline().strip().split("\t")[1:]
    for line in f:
        row = line.strip().split("\t")
    return data
```

Sedaj moramo to samo še zapakirat v neko strukturo, recimo slovar. Ključ bo kar ime osebe, torej prvi element v seznamu, preostanek seznama pa so ocene, dejanski podatki, ki jih potrebujemo za primerjavo. Ne smemo pozabit, da iz datoteke beremo podatke kot nize, zato jih moramo še spremenit v številke. To lahko naredimo z enostavno for zanko.

```
numbers = []
for n in row[1:]:
    numbers.append(float(n))
```

Ampak Python zna takšne enostavne for zanke tudi lepše napisat. Načinu se reče list comprehension

```
numbers = [float(n) for n in row[1:]]
```

Tega seveda ne rabite uporabljat. Osebno se mi zdi, da je koda tako veliko bolj jasna, dejansko pa je tudi kanček hitrejša, saj se čas izvajanja zmanjša za čas klica append funkcije.

Končno združimo to vse v funkcijo, ki prebere podano datotek ter shrani podatke v slovar in ga vrne. To je sicer malo banalen primer ampak v splošnem vam priporočam, da najprej spišete košček programa v lupini, jih testirate in nato združite v celoto.

```
def read_file(file_name):
    with open(file_name) as f:
        header = f.readline().strip().split("\t")[1:]
        data = {}
        for line in f:
            row = line.strip().split("\t")
            data[row[0]] = [float(x) for x in row[1:]]
    return data
```

Sedaj ko smo sprogramirali prvo večjo funkcijo je čas, da si pogledamo najpomembnejši aspekt programiranja – razhroščevanje. Poglejmo kako se to pravilno počne v PyCharm-u. Nastavim točko zaustavitve. Poženem razhroščevalnik, pokažem kako se premikaš po kodi, pokažem izpis spremenljivke, pokažem kako vstopit v interaktivno lupino in kako spremembe vplivajo na izvajanje programa.

Nadaljujem z izpisom komentarjev v razred in posamezne metode. Pojasnim, da je __init__ metoda funkcija, ki se požene ob inicializaciji objekta. Pojasnim, da vse metode kot prvi argument prejmejo spremenljivko self v kateri se nahaja instanco objekta. Pri inicializaciji nastavimo skupine v katerih so po en primer. Zopet uporabimo list comprehentin. Shranimo v spremenljivko self.clusters.

Ok, sedaj začnimo pri najenostavnejši funkciji – row_distance. Želimo si, da bi jo lahko poklicali kot: self.row_distance("Polona", "Rajko"), ta pa nam bi vrnila evklidsko razdaljo med vektorjema ocen. Pa se spomnimo na hitro kaj je evklidska razdalja. Ok, torej vem že kako bomo dobili dva seznama ocen iz slovarja. Sedaj moremo samo sprogramirat del, ki iz dveh seznamov (ali vektorjev)

izračuna razdaljo. Da nam bo lažje si zopet najprej pomagajmo s interaktivno lupino. Lahko bi šli po c-jevsko in v for zanki povečevali števec ter tako preko indeksa dobili posamezne elemente. Mogoče se da v Pythonu to kako lepše naredit. Poggoglaj 'python pair elements from two lists'

```
xs = [1, 2, 3]
ys = [2, 3, 4]
s = []
for x, y in zip(xs, ys):
    s.append((x – y) ** 2)
```

Aha zopet smo uporabil zgolj append, to sedaj znamo že krajše napisat:

```
s = [(x – y) ** 2 for x, y in zip(xs, ys)]
```

Sedaj moremo to vse samo seštet in korenit. Sumim, da bi morala obstajat funkcija, ki bi seštela seznam in izračunala koren. Res je med build-in funkcijami sum(), korena pa ne najdem. Pogooglajmo 'python square'. Očitno obstaja v knjižnici math funkcija sqrt, ki počne ravno to. No natančneje knjižnicam rečemo moduli. Nekaj je takšnih, ki že pridejo skupaj z Pythonom. Pogooglajmo 'python buildin modules'. Modul lahko uvozimo na več načinov.

```
import math
from math import sqrt
```

No pa združimo kar smo do sedaj naredili. Dejansko lahko zapišemo celotno kodo v eni vrstici. Pri klicu funkcije sum ne rabimo dodat oglatih oklepajev. Za tiste, ki jih zanima zakaj si naj pogledajo kaj so generatorji v Pythonu.

```
math.sqrt(sum((x – y) ** 2 for x, y in zip(xs, ys)))
```

Sedaj moremo v svojem programu samo še zamenjat xs in ys z vektorji iz slovarja.

Funkcijo cluster_distance boste morali spisat sami. Tukaj boste morali najprej razstavit gnezden seznam seznamov v en sam seznam elementov nato pa po formuli izračunat želen linkage.
Prav tako boste morali sami napisat funkcijo run in plot_tree.

Sedaj bomo na hitro spisali samo še closest_clusters. Se pravi najti moramo tak par skupin, ki so si najbližje. Drugače povedano izračunati moremo razdaljo med vsemi kombinacijami trenutnih skupin, ter izmed njih izbrat tak par, katere razdalja je najmanjša. Ok, začnimo pri kombinacijah, to se sliši kot nekaj, za kar bi morala že obstajat funkcija, poglejmo kaj pravi google. Uporabimo combinations funkcijo iz modula itertools. Ta nam vrača terke ki jih podobno kot prej razpakiramo vsako v svojo spremenljivko. Med njimi izračunamo razdaljo, za to že imamo funkcijo.

```
[cluster_distance(c1, c2) for c2, c2 in combinations(clusters, 2)]
```

Očitno moremo tej vrednosti shranit, saj bomo morali kasneje najt najmanjšo.
Ker pa nas ne zanima samo najkrajša razdalja ampak tudi par skupin, ki ima to razdaljo si moramo shranit oboje. Prav v takih primerih se ponavadi uporabljajo terke. Torej naredimo:

```
[(cluster_distance(c1, c2), (c1, c2)) for c2, c2 in combinations(clusters, 2)]
```

Sedaj moramo najt najmanjši element v seznamu. Python pri urejanju terk upošteva najprej prve elemente nato druge in tako naprej. Ker že vemo, da ima Python bogat nabor standardnih funkcij smo prepričani, da mora obstajat funkcija min. In res da :). Za lažje razumevanje nazaj odpakirajmo končno vrednost.

dis, pair = min((cluster_distance(c1, c2), (c1, c2)) for c2, c2 in combinations(clusters, 2))

Ok to je vse, kar vam lahko pokažem od domače naloge. Dolžan pa sem vam še nekaj pojasnitev. Nismo si namreč še ogledali testov. Odprem datoteko in pokažem teste. Razložim, da je vsaka metoda, ki se začne z besedo test eden od testov. Pojasnim, da se znotraj testa ponavadi izvede del kode ter s pomočjo assert stavkov preveri pravilnost izhodov. Poizkusim pognat in ne dela :(Oh očitno testi uporabljajo zunanje knjižnice. Tej je potrebno dodatno inštalirat. Tukaj se morem vrnit tudi nazaj na virtualna okolja. To so kopije Python interpreterja skupaj z dodatno inštaliranimi knjižnicami. Tako kadar inštaliraš dodaten modul se ta namesti samo v to določeno okolje. Obstaja več razlogov zakaj jih je dobro uporabljat glavni izmed katerih je, da ne onesnažujemo sistemske inštalacije Pythona. Dobra praksa je, da imaš ločene inštalacije Pythona za vsak projekt na katerem delaš. Na spletni strani pypi lahko najdete ogromen nabor paketov za Python. Paketi se namestijo s pomočjo orodja pip. Pozor, vsako virtualno okolje ima svoj pip. Nekateri paketi imajo del kode implementiran v C-ju in ta se ob inštalaciji prevede. Tako lahko nastanejo problemi, če nimate potrebnih programov za prevajat. Windows uporabniki imate lahko še posebej probleme. Za vas v takih primerih priporočam uporabo programa miniconda, ki omogoča inštalacijo že prevedenih knjižnic.

Sedaj, ko smo inštalirali naše dodatne module lahko končno poženemo teste. Pokažem grafičen vmesnik testov.