5 Klase i interfejsi

Kao objektno orijentisan programski jezik, Python podržava pun opseg mogućnosti, kao što su nasleđivanje, polimorfizam i kapsuliranje. Da bi se stvari završile u Pythonu, često je potrebno napisati nove klase i definisati njihove interakcije kroz interfejse i hijerarhije klasa.

Pythonove klase i nalseđivanje olakšavaju izražavanje željenog ponašanja programa pomoću objekata. Oni omogućavaju da vremenom poboljšate i proširite funkcionalnost. Oni nude fleksibilnost kada se zahtevi menjaju. Ako znate kako da ih koristite, moći ćete da pišete održivi kod.

Tema 37: Spajajte klase umesto da ugnežđujete više nivoa ugrađenih tipova

Pythonov ugrađeni tip rečnika divan je za održavanje dinamičkog unutrašnjeg stanja tokom životnog veka nekog objekta. Kada kažem *dinamičkog*, mislim na situacije u kojima treba da vodite evidenciju neočekivanog skupa identifikatora. Na primer, recimo da želim da evidentiram ocene skupa studenata čija mi imena nisu unapred poznata. Mogu da definišem klasu da bih skladištio imena u rečniku, umesto da koristim unapred definisan atribut za svakog studenta:

```
class SimpleGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)

    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)
```

Korišćenje klase je jednostavno:

```
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
book.report_grade('Isaac Newton', 95)
book.report_grade('Isaac Newton', 85)

print(book.average_grade('Isaac Newton'))

>>>
90.0
```

Rečnici i s njima povezani ugrađeni tipovi toliko se lako koriste da postoji opasnost njihovog preteranog proširivanja i dobijanja krhkog koda. Recimo da želim da proširim klasu `SimpleGradebook` tako da čuva listu ocena po predmetu, a ne samo ukupnih ocena. Da bih to uradio, mogu da izmenim rečnik `_grades` tako da uparuje imena studenata (ključeve rečnika) sa još jednim rečnikom (vrednostima). Unutrašnji rečnik će uparivati predmete (svoje ključeve) sa listom ocena (vrednostima). Ovde to radim koristeći instancu `defaultdict` za unutrašnji rečnik kojim rešavam problem nedostajućih predmeta (Tema 17: Za nedostajuće stavke u unutrašnjem stanju klase dajte prednost klasi `defaultdict` u odnosu na `setdefault`):

```
from collections import defaultdict

class BySubjectGradebook:
    def __init__(self):
        self._grades = {}                       # Spoljni rečnik

    def add_student(self, name):
        self._grades[name] = defaultdict(list)  # Unutrašnji r.
```

Ovo deluje jasno. Metode `report_grade` i `average_grade` postaju prilično složene da bi se izborile sa više nivoa rečnika, ali čini se da je to podnošljivo:

```
    def report_grade(self, name, subject, grade):
        by_subject = self._grades[name]
        grade_list = by_subject[subject]
        grade_list.append(grade)

    def average_grade(self, name):
        by_subject = self._grades[name]
        total, count = 0, 0
        for grades in by_subject.values():
```

```
            total += sum(grades)
            count += len(grades)
        return total / count
```

Korišćenje klase i dalje je jednostavno:

```
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
print(book.average_grade('Albert Einstein'))
```

```
>>>
81.25
```

Zamislite sad da se zahtevi ponovo promene. Želim da pratim i težinu svake ocene u konačnoj oceni, tako da ispiti na polugodištu i kraju godine imaju veći značaj nego nenajavljeni testovi. Jedan način da implementiram tu mogućnost jeste da izmenim unutršanji rečnik; umesto da uparuje predmete (svoje ključeve) sa listom ocena (vrednostima), mogu da koristim n-torku koju čini ocena i njena težina (`score`, `weight`) u listi vrednosti:

```
class WeightedGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = defaultdict(list)

    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject[subject]
        grade_list.append((score, weight))
```

Iako se čini da su izmene metode `report_grade` jednostavne – samo sam zadao da lista ocena skladišti instance n-torke – metoda `average_grade` sada ima petlju unutar petlje i teško se čita:

```
    def average_grade(self, name):
        by_subject = self._grades[name]

        score_sum, score_count = 0, 0
        for subject, scores in by_subject.items():
            subject_avg, total_weight = 0, 0
```

```
            for score, weight in scores:
                subject_avg += score * weight
                total_weight += weight

            score_sum += subject_avg / total_weight
            score_count += 1

        return score_sum / score_count
```

I korišćenje klase je postalo teže. Nije jasno šta znače svi oni brojevi u pozicionim argumentima:

```
book = WeightedGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75, 0.05)
book.report_grade('Albert Einstein', 'Math', 65, 0.15)
book.report_grade('Albert Einstein', 'Math', 70, 0.80)
book.report_grade('Albert Einstein', 'Gym', 100, 0.40)
book.report_grade('Albert Einstein', 'Gym', 85, 0.60)
print(book.average_grade('Albert Einstein'))

>>>
80.25
```

Kada vidite ovako složen kod, vreme je da sa ugrađenih tipova kao što su rečnici, n-torke, skupovi i liste, pređete na hijerarhiju klasa.

U primeru sa ocenama, na početku nisam znao da ću morati da programiram podršku za težinu ocena, pa se složeno pravljenje klasa činilo neopravdanim. Pythonovi ugrađeni tipovi rečnika i n-troke olakšali su nastavljanje rada i dodavanje sve više slojeva unutrašnjoj evidenciji. Trebalo bi da to izbegavate kada imate više od jednog nivoa ugnežđivanja; korišćenje rečnika koji sadrže rečnike otežava drugim programerima čitanje koda i smeštaju vas u noćnu moru održavanja.

Čim shvatite da evidencija postaje složena, podelite je u klase. Tada možete da obezbedite dobro definisane interfejse koji bolje kapsuliraju podatke. Taj pristup omogućava i da napravite sloj apstrakcije između interfejsa i konkretnih implementacija.

Prerađivanje u klase

Postoji mnogo pristupa prerađivanju (Tema 89: Razmotrite `warnings` u refaktorisanju i migraciji, opisuje još jedan). U ovom slučaju mogu da počnem prelazak na klase od dna stabla zavisnosti: od jedne ocene. Čini se da je klasa preterano kolosalna za tako jednostavnu informaciju. S druge strane, n-torka deluje odgovarajuće jer su ocene neizmenjive. Ovde koristim n-torku sačinjenu od ocene i njene težine (`score, weight`) da bih pratio ocene u listi:

```
grades = []
grades.append((95, 0.45))
grades.append((85, 0.55))
total = sum(score * weight for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

Upotrebio sam _ (ime promenljive s podvlakom, Pythonova konvencija za nekorišćene promenljive) da bih uhvatio prvi ulaz u n-torki svake ocene i zanemario ga kada budem izračunavao ukupnu težinu, `total_weight`.

Problem sa ovim kodom je u tome što su instance n-torki pozicione. Na primer, ako želim da povežem više informacija sa ocenom, na primer, skup napomena nastavnika, treba da prepravim svako korišćenje dvočlane n-torke da bih znao da sada postoje tri stavke, a ne dve, što znači da moram i dalje da koristim _ da bih zanemario izvesna indeksna mesta:

```
grades = []
grades.append((95, 0.45, 'Great job'))
grades.append((85, 0.55, 'Better next time'))
total = sum(score * weight for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

Ovakav obrazac produžavanja n-torki sličan je produbljivanju slojeva u rečnicima. Čim primetite da ste otišli dalje od dvočlane n-torke, vreme je da razmislite o drugom pristupu.

Tip `namedtuple` (imenovana n-torka) iz ugrađenog modula `collections` radi upravo ono što mi je u ovom slučaju potrebno: omogućava mi da lako definišem malecke, neizmenjive klase podataka:

```
from collections import namedtuple

Grade = namedtuple('Grade', ('score', 'weight'))
```

Te klase se mogu konstruisati sa pozicionim argumentima ili argumentima po imenu. Polja su dostupna preko imenovanih atributa. Imenovani atributi olakšavaju kasniji prelazak sa tipa `namedtuple` na klasu, ukoliko se zahtevi ponovo promene i ako budem morao, na primer, da dodam podršku za izmenjivost ili ponašanja u jednostavnim kontejnerima podataka.

Ograničenja tipa `namedtuple`

Iako je `namedtuple` koristan tip u mnogim okolnostima, bitno je razumeti kada on može da napravi više štete nego koristi:

- Ne možete unapred zadati vrednosti argumenata za klase `namedtuple`. To ih čini nezgrapnim kada podaci imaju više opcionih svojstava. Ukoliko koristite više od šačice atributa, možda bi bilo bolje da upotrebite ugrađeni modul `dataclasses`.
- Vrednostima atributa u instancama `namedtuple` i dalje se može pristupati korišćenjem numeričkih indeksa i iteracije. Naročito u spoljnim API-ima, to može dovesti do slučajnog korišćenja koje kasnije otežava prelazak na pravu klasu. Ako ne upravljate samo vi svakim korišćenjem instanci `namedtuple`, bolje je da izričito definišete novu klasu.

Potom mogu da napišem klasu koja će predstavljati jedan predmet i sadržati skup ocena:

```
class Subject:
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

Zatim pišem klasu koja predstavlja skup predmeta koje jedan student uči:

```
class Student:
    def __init__(self):
        self._subjects = defaultdict(Subject)

    def get_subject(self, name):
        return self._subjects[name]
```

```
    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```

Na kraju bih napisao kontejner za sve studente, sa dinamičkim ključevima u vidu njihovih imena:

```
class Gradebook:
    def __init__(self):
        self._students = defaultdict(Student)

    def get_student(self, name):
        return self._students[name]
```

Broj redova za ove klase gotovo je dvostruko veći od prethodne implementacije. Međutim, ovaj kod se mnogo lakše čita. Primer u kojem se pokreću klase takođe je jasniji i proširiv je:

```
book = Gradebook()
albert = book.get_student('Albert Einstein')
math = albert.get_subject('Math')
math.report_grade(75, 0.05)
math.report_grade(65, 0.15)
math.report_grade(70, 0.80)
gym = albert.get_subject('Gym')
gym.report_grade(100, 0.40)
gym.report_grade(85, 0.60)
print(albert.average_grade())

>>>
80.25
```

Bilo bi moguće napisati i metode kompatibilne sa starijim verzijama da bi se lakše prešlo sa korišćenja starog stila API na novu hijerarhiju objekata.

Zapamtite

- Izbegavajte pravljenje rečnika sa vrednostima koje su rečnici, dugačke n-torke ili drugi ugrađeni tipovi koji su složeno ugnežđeni.
- Koristite tip `namedtuple` za lake, neizmenjive kontejnere podataka pre nego što vam bude potrebna fleksibilnost pune klase.
- Kod evidencije prebacite na korišćenje više klasa kada rečnici unutrašnjeg stanja postanu previše složeni.