POGLAVLJE 8

Klase i objekti

Osnovni fokus ovog poglavlja je predstavljanje recepata za uobičajene šeme programiranja vezane za definicije klasa. Između ostalog, govorimo o podršci objekata za uobičajene Pythonove alatke, primeni specijalnih metoda, tehnikama kapsuliranja, nasleđivanju, upravljanju memorijom i korisnim šemama projektovanja.

8.1. Izmena formata instanci predstavljenih u vidu znakovnih nizova

Problem

Hoćete da izmenite rezultat ispisivanja ili pregledanja instanci u nešto prigodnije.

Rešenje

Da biste instancu, umesto kao znakovni niz, predstavili u drugom obliku, definišite metode `__str__()` i `__repr__()`. Na primer:

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

Metoda `__repr__()` vraća detalje koda za predstavljanje instance, a to je obično tekst koji biste upisali da ponovo napravite instancu. Ugrađena funkcija `repr()` vraća ovaj tekst, kao i interaktivni interpreter prilikom ispitivanja vrednosti. Metoda `__str__()` konvertuje instancu u znakovni niz, što je rezultat koji vraćaju funkcije `str()` i `print()`. Na primer:

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4)          # Rezultat metode __repr__()
>>> print(p)
(3, 4)              # Rezultat metode __str__()
>>>
```

Ovaj recept pokazuje i kako se različiti načini predstavljanja znakovnog niza mogu upotrebiti prilikom formatiranja. Konkretno, kôd formata `!r` označava da bi podrazumevano trebalo da koristi rezultat metode `__repr__()`, a ne metode `__str__()`. Mogli biste da napravite eksperiment s prethodnom klasom:

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>
```

Objašnjenje

Definisanje metoda `__repr__()` i `__str__()` često je dobra praksa, jer može da pojednostavi otkrivanje i otklanjanje grešaka i prikazivanje instance. Na primer, dovoljno je da se instanca ispiše ili evidentira, pa da programer dobije dodatne korisne informacije o sadržaju instance.

Uobičajeno je da metoda `__repr__()` generiše tekst kao što je `eval(repr(x)) == x`. Ukoliko to nije moguće ili nije željeno ponašanje, obično se umesto toga pravi korisna tekstualna reprezentacija između oznaka `<` i `>`. Na primer:

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

Ukoliko metoda `__str__()` nije definisana, kao zamena se koristi rezultat metode `__repr__()`.

Primena funkcije `format()` u rešenju možda je pomalo neočekivana, ali kôd formata `{0.x}` zadaje atribut `x` argumenta 0. Zato je, u narednoj funkciji, `0` zapravo instanca `self`:

```
def __repr__(self):
    return 'Pair({0.x!r}, {0.y!r})'.format(self)
```

Umesto ovakve implementacije, mogli ste da primenite operator `%` i naredni kôd:

```
def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)
```

8.2. Prilagođavanje formatiranja znakovnog niza

Problem

Želite da objekat podržava prilagođeno formatiranje pomoću funkcije i metode znakovnih nizova `format()`.

Rešenje

Da biste prilagodili formatiranje znakovnih nizova, definišite metodu `__format__()` u klasi. Na primer:

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
    }

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

Instance klase `Date` sada podržavaju operacije formatiranja poput narednih:

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

Objašnjenje

Metoda `__format__()` je spona sa Pythonovom funkcionalnošću za formatiranje znakovnih nizova. Važno je znati da tumačenje kodova formatiranja potpuno zavisi od same klase. Dakle, kodovi mogu biti gotovo bilo šta. Na primer, razmotrimo naredni kôd s modulom `datetime`:

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d,'%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

Postoje neke standardne konvencije za formatiranje ugrađenih tipova. Formalnu specifikaciju naći ćete u dokumentaciji za modul `string` (*http://docs.python.org/3/library/string.html*).

8.3. Uspostavljanje podrške objekata za protokol upravljanja kontekstom

Problem

Želite da vaši objekti podržavaju protokol upravljanja kontekstom (naredba `with`).

Rešenje

Da bi objekat bio kompatibilan s naredbom `with`, morate implementirati metode `__enter__()` i `__exit__()`. Na primer, razmotrimo narednu klasu koja omogućava mrežnu vezu:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

Ključna odlika ove klase jeste to što predstavlja mrežnu vezu, ali ne radi inicijalno ništa sama (recimo, ne uspostavlja vezu). Umesto toga, veza se uspostavlja i zatvara pomoću naredbe `with` (u suštini, po zahtevu). Na primer:

```
from functools import partial

conn = LazyConnection(('www.python.org', 80))
# Veza je zatvorena
with conn as s:
    # conn.__enter__() se izvršava: veza otvorena
    s.send(b'GET /index.html HTTP/1.0rn')
    s.send(b'Host: www.python.orgrn')
    s.send(b'rn')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() se izvršava: veza zatvorena
```

Objašnjenje

Kada pišete upravljač kontekstom, imajte na umu da pišete kôd koji treba da okruži blok naredaba definisanih putem naredbe `with`. Kada se u izvršavanju dođe do naredbe `with`,

poziva se metoda `__enter__()`. Povratna vrednost metode `__enter__()` (ako je ima) smešta se u promenljivu na koju ukazuje kvalifikator `as`. Posle toga, izvršavaju se naredbe u telu naredbe `with`. Na kraju se poziva metoda `__exit__()` da počisti.

Ovakav tok izvršavanja važi bez obzira na to šta se dešava u telu naredbe `with`, pa i ako dođe do izuzetaka. Zapravo, tri argumenta metode `__exit__()` sadrže tip izuzetka, vrednost i izveštaj o praćenju steka (engl. *traceback*) za izuzetke na čekanju (ako ih ima). Metoda `__exit__()` može da odabere da li da iskoristi informacije o steku na neki način ili da ih zanemari tako što neće ništa uraditi sem što će vratiti `None` kao rezultat. Ako metoda `__exit__()` vrati `True`, izuzetak se briše kao da se ništa nije desilo i program nastavlja da izvršava naredbe neposredno iza bloka `with`.

Jedno od manje očiglednih pitanja u vezi sa ovim receptom jeste da li klasa `LazyConnection` dozvoljava ugnežđenu primenu veze s većim brojem `with` naredaba. Kako je prikazano, samo jedna veza utičnice dozvoljena je u datom trenutku, a ukoliko se naredba `with` i dalje poziva dok je utičnica u upotrebi, javlja se izuzetak. Ovo ograničenje možete premostiti uz male izmene u implementaciji:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.connections = []

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.connections.pop().close()

# Primer primene
from functools import partial

conn = LazyConnection(('www.python.org', 80))
with conn as s1:
     ...
     with conn as s2:
          ...
          # s1 i s2 su nezavisne utičnice
```

U drugoj verziji, klasa `LazyConnection` je svojevrsna fabrika veza. Interno, za stek se koristi lista. Kad god se `__enter__()` izvršava, uspostavlja se nova veza i dodaje se steku. Metoda `__exit__()` skida poslednju vezu sa steka i zatvara je. Ovo rešenje omogućava da se napravi više veza odjednom pomoću ugnežđenih naredaba `with`, kao što je prikazano.

Upravljači kontekstom se često koriste u programima koji upravljaju resursima kao što su datoteke, mrežne veze i brave. Najvažniji aspekt je to što se takvi resursi moraju eksplicitno zatvoriti ili osloboditi da bi se program valjano izvršavao. Na primer, ako napravite bravu, morate je i otpustiti, tj. osloboditi memoriju, inače rizikujete kružnu blokadu (engl. *deadlock*). Ako implementirate metode `__enter__()`, `__exit__()` i primenite naredbu `with`, mnogo lakše ćete izbeći takve probleme, pošto će se kôd za čišćenje u metodi `__exit__()` u svakom slučaju izvršiti.

Alternativna formulacija upravljača kontekstom može se naći u modulu `contextmanager`. Videti Recept 9.22. Verzija ovog recepta bezbedna za višenitni rad dostupna je u Receptu 12.6.

8.4. Ušteda u memoriji prilikom pravljenja velikog broja instanci

Problem

Vaš program pravi mnogo (na primer, milione) instanci, i pritom koristi ogromnu količinu memorije.

Rešenje

Za klase koje primarno služe kao jednostavne strukture podataka, često je moguće ostvariti znatnu uštedu u memoriji dodavanjem deklaracije `__slots__`. Na primer:

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

Kada definišete `__slots__`, Python koristi mnogo sažetiju internu prezentaciju instanci. Umesto od rečnika, instance se, u osnovi, prave od malog niza fiksne veličine, nalik n-torci ili listi. Imena atributa navedena u specifikatoru `__slots__` interno se mapiraju na određene indekse u okviru tog niza. Sporedna posledica ovakvog rešenja je to što više nije moguće dodavati nove atribute instanci – ograničeni ste samo na imena atributa navedena u specifikatoru `__slots__`.

Objašnjenje

Memorija koja se uštedi zahvaljujući slotovima zavisi od broja i tipa sačuvanih atributa. Međutim, načelno, upotrebljena memorija je porediva s memorijom za skladištenje podataka u n-torci. Da biste stekli predstavu, pomenimo da čuvanje jedne instance tipa `Date` bez slotova zahteva 428 bajta memorije na 64-bitne verzije Pythona. Kada se definišu slotovi, zauzeta memorija se smanjuje na 156 bajtova. U programu koji istovremeno upravlja velikim brojem datuma, na ovaj način bi se ostvarila značajna ušteda u ukupnoj upotrebljenoj memoriji.

Iako se slotovi možda čine kao nešto što bi uvek moglo biti korisno, oduprite se iskušenju da im prečesto pribegavate. Mnogi delovi Pythona se oslanjaju na normalnu implementaciju zasnovanu na rečniku. Pored toga, klase koje ne definišu slotove ne podržavaju određene mehanizme (na primer, višestruko nasleđivanje). U većini slučajeva, slotove bi trebalo da koristite samo za klase koje će se često koristiti u programu kao strukture podataka (to jest, ako vaš program pravi milione instanci određene klase).

Često se za specifikator `__slots__` pogrešno misli da predstavlja alatku kapsuliranja koja sprečava korisnike da dodaju nove atribute instancama. Iako to jeste sporedna posledica primene slotova, to nikada nije bila prvobitna namera. Umesto toga, zamisao je bila da `__slots__` bude alatka za optimizaciju.

8.5. Kapsuliranje imena u klasi

Problem

Želite da kapsulirate „privatne“ podatke instance klase, ali brine vas Pythonov nedostatak kontrole pristupa.

Rešenje

Umesto da se oslanjaju na elemente jezika za kapsuliranje podataka, Pythonovi programeri bi trebalo da imaju na umu određene konvencije imenovanja u vezi sa željenom upotrebom podataka i metoda. Prva konvencija je da za svako ime koje počinje jednom donjom crtom (_) uvek treba da pretpostave da označava internu implementaciju. Na primer:

```
class A:
    def __init__(self):
        self._internal = 0    # Interni atribut
        self.public = 1       # Javni atribut

    def public_method(self):
        '''
        A public method
        '''
        ...

    def _internal_method(self):
        ...
```

Python vas, zapravo, ne sprečava da pristupate internim imenima. Ipak, to se smatra nepristojnim, i moglo bi da dovede to nestabilnog koda. Valjalo bi istaći i da se vodeća donja crta koristi i za imena modula i funkcije na nivou modula. Na primer, ako naiđete na ime modula koje počinje donjom crtom (kao što je `_socket`), znajte da je reč o internoj implementaciji. Slično tome, funkcije na nivou modula kao što je `sys._getframe()` koristite vrlo oprezno.

Možda ćete se susresti i s dve početne donje crte (__) u imenima u okviru definicija klase. Na primer:

```
class B:
    def __init__(self):
        self.__private = 0
    def __private_method(self):
        ...
    def public_method(self):
        ...
        self.__private_method()
        ...
```

Zbog prefiksa u vidu duple donje crte, ime je iseckano. Konkretno, privatni atributi u prethodnoj klasi preimenuju se u `_B__private` i `_B__private_method`. Možda se sada pitate čemu takvo seckanje imena. Odgovor je u nasleđivanju – ovakvi atributi se ne mogu redefinisati putem nasleđivanja. Na primer:

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1      # Ne redefiniše B.__private
    # Ne redefiniše B.__private_method()
    def __private_method(self):
        ...
```

U ovom primeru, privatna imena `__private` i `__private_method` izmenjena su u `_C__private` i `_C__private_method` – dakle, drugačija su od imena u osnovnoj klasi `B`.

Objašnjenje

Činjenica da postoje dve konvencije (s jednom, odnosno dve donje crte) za privatne atribute, navodi da se zapitate koji stil bi trebalo da primenite. U većini slučajeva, verovatno bi imena koja nisu javna trebalo da počinjete jednom donjom crtom. Međutim, ako znate da će vaš kôd da dovede do potklasa i postoje interni atributi koji bi trebalo da budu sakriveni od potklasa, primenite umesto toga duplu donju crtu.

Valjalo bi istaći i da ćete ponekad možda hteti da definišete promenljivu istog imena kao rezervisana reč. U tom slučaju, trebalo bi da stavite jednu prateću donju crtu. Na primer:

```
lambda_ = 2.0     # Prateća _ da bi se izbeglo poklapanje s rezervisanom rečju lambda
```

U ovom slučaju, početnu donju crtu nismo primenili da bismo izbegli zabunu u vezi s namerom (to jest, vodeća donja crta mogla bi se protumačiti kao način da se predupredi konflikt imena, umesto kao indikacija da je vrednost privatna). Prateća donja crta rešava taj problem.

8.6. Pravljenje upravljanih atributa

Problem

Hoćete da proširite obradu (na primer, da dodate proveru tipa ili ispravnosti) u okviru čitanja ili zadavanja atributa instance.