9.1. (Ne)ponořme se

Ta dnešní mládež. Jsou tak zkažení těmi rychlými počítači a módními „dynamickými“ jazyky. Rychle napsat, pak dodat a ladit až nakonec (jestli vůbec). Za mých časů jsme dodržovali disciplínu. Říkám disciplínu! Museli jsme psát programy *ručně*, na *papír* a cpát je do počítače na *děrných štítcích*. A ono se nám to *líbilo*! A cože? Že je ten nadpis anglicky? Buďte rádi, že není v ruštině. Mnozí z vás ani neví, jak přečíst jednotlivá písmenka azbuky. No dobrá, trochu zvážním. Dá se to přeložit jako „testování jednotek“ nebo „jednotkové testování“. Ještě se k tomu dostaneme.

V této kapitole si napíšeme a odladíme pár pomocných funkcí pro konverzi na a z římských čísel. Způ-
Kap. sob tvorby a ověřování římských čísel jsme si ukázali v podkapitole Případová studie: Římská čísla. Teď si poodstoupíme a zvážíme, kolik by dalo práce rozšířit původní kód na obousměrné pomocné funkce.

Pravidla pro římská čísla vedla k řadě zajímavých postřehů:

1. Existuje jen jeden správný způsob vyjádření konkrétního čísla římskými číslicemi.
2. Platí také opak. Pokud je řetězec znaků platným římským číslem, reprezentuje jen jedno možné číslo (to znamená, že řetězec může být interpretován jen jedním způsobem).
3. Římskými čísly lze vyjádřit jen omezený rozsah čísel, konkrétně od `1` do `3999`. Římané používali několik způsobů vyjádření větších čísel. Tak například pruhem nad římským číslem vyjadřovali, že jeho číselná hodnota musí být vynásobená tisícem. Pro účely této kapitoly budeme uvažovat jen římská čísla od `1` do `3999`.
4. Neexistuje způsob, jak římskými číslicemi vyjádřit nulu.
5. Neexistuje způsob, jak římskými číslicemi vyjádřit záporná čísla.
6. Neexistuje způsob, jak římskými číslicemi vyjádřit zlomky nebo neceločíselné hodnoty.

Začněme mapovat, co by takový modul roman.py měl dělat. Bude obsahovat dvě hlavní funkce, `to_roman()` (na římské číslo) a `from_roman()` (z římského čísla). Funkce `to_roman()` by měla převzít celé číslo v intervalu od `1` do `3999` a vrátit jeho reprezentaci římskými číslicemi jako řetězec...

Hned tady se zastavíme. Teď uděláme něco trošku neočekávaného. Napíšeme si testovací příklad, který kontroluje, zda funkce `to_roman()` dělá to, co po ní chceme. Čtete dobře. Jdeme psát kód, který testuje jiný kód, který jsme ještě nenapsali.

Říká se tomu *vývoj řízený testy (test-driven development)* nebo `TDD`. (V anglické literatuře si potrpí na zavádění a používání zkratek.) Dvojice převodních funkcí — `to_roman()` a později `from_roman()` — může být napsána a testována jako *jednotka* (unit), odděleně od jakéhokoliv většího programu, který funkce importuje. V Pythonu najdeme rámec (framework) pro unit testing (tedy testování jednotek), který má podobu příhodně nazvaného modulu `unittest`.

Unit testing (testování jednotek) představuje důležitou součást celkové vývojové strategie založené na testování. Pokud testy jednotek píšete, je důležité, abyste je napsali brzy a abyste je udržovali v závislosti na změnách kódu a požadavků. Mnozí lidé se přimlouvají za to, aby se testy psaly dříve

než kód, který mají testovat. V této kapitole si takový přístup předvedeme. Ale testy jednotek mají své výhody nezávisle na tom, kdy je napíšete.

- Napsání jednotkových testů (i takto se to dá překládat) ještě před napsáním kódu vás účelným způsobem donutí upřesnit své požadavky
- Při vlastním psaní kódu vás pak jednotkové testy brzdí před psaním nadbytečných věcí. Jakmile všechny testy projdou, dosáhli jste úplné funkčnosti.
- Při provádění refaktorizace kódu vám testy jednotek pomohou prokázat, že se nová verze chová stejným způsobem jako ta stará.
- Při údržbě kódu vám existence testů pomůže krýt záda (v originále se mluví o té části těla, kde záda ztrácejí své slušné jméno) v situaci, kdy na vás někdo přiletí a řve, že vaše poslední změny pokazily jejich původní kód. („Ale *pane*, ale když jsem změny zveřejňoval, všechny unit testy prošly...“)
- Pokud píšeme kód v týmu, pak existence společné sady testů dramaticky snižuje možnost, že by váš kód způsobil nefunkčnost kódu někoho jiného. Jejich testy jednotek totiž můžete spustit jako první. (Tenhle druh závodů v psaní kódu už jsem zažil. Tým si zadání rozdělí, každý si převezme specifikace svého úkolu, napíše pro něj jednotkové testy a pak je dá k dispozici ostatním členům týmu. Při takovém postupu nikdo nezabloudí tak daleko, že by jím vyvíjený kód nespolupracoval s výsledky ostatních.)

9.2. Jediná otázka

Každý test je ostrov.

Testovací případ (test case) odpovídá na jedinou otázku, která se testovaného kódu týká. Testovací případ by měl být schopen...

- ...běžet zcela samostatně, bez jakéhokoliv lidského zásahu. Unit testing (testování jednotek) souvisí s automatizací.
- ...sám rozhodnout o tom, zda testovaná funkce prošla nebo selhala — bez nutnosti posuzování výsledků člověkem.
- ...běžet izolovaně, odděleně od jakýchkoliv jiných testovacích případů (dokonce i když testují stejnou funkci). Každý testovací případ je ostrov.

S ohledem na uvedené předpoklady začněme budovat testovací případ pro první požadavek:

> Funkce `to_roman()` by měla vracet reprezentaci římského čísla pro všechna celá čísla v intervalu `1` až `3999`.

V prvním okamžiku není zřejmé, jak následující kód dělá... no vlastně *cokoliv*. Definuje třídu, která nemá žádnou metodu `__init__()`. Třída sice *má* nějakou metodu, ale ta se nikdy nevolá. Celý skript obsahuje blok `__main__`, ale nenajdeme v něm odkaz ani na třídu, ani na její metodu. Ale on opravdu něco dělá. Za to ručím.

```
import roman1
import unittest

class KnownValues(unittest.TestCase):                                    [1]
    known_values = ( (1, 'I'),
                     (2, 'II'),
                     (3, 'III'),
                     (4, 'IV'),
                     (5, 'V'),
                     (6, 'VI'),
                     (7, 'VII'),
                     (8, 'VIII'),
                     (9, 'IX'),
                     (10, 'X'),
                     (50, 'L'),
                     (100, 'C'),
                     (500, 'D'),
                     (1000, 'M'),
                     (31, 'XXXI'),
                     (148, 'CXLVIII'),
                     (294, 'CCXCIV'),
                     (312, 'CCCXII'),
                     (421, 'CDXXI'),
                     (528, 'DXXVIII'),
                     (621, 'DCXXI'),
                     (782, 'DCCLXXXII'),
                     (870, 'DCCCLXX'),
                     (941, 'CMXLI'),
                     (1043, 'MXLIII'),
                     (1110, 'MCX'),
                     (1226, 'MCCXXVI'),
                     (1301, 'MCCCI'),
                     (1485, 'MCDLXXXV'),
                     (1509, 'MDIX'),
                     (1607, 'MDCVII'),
                     (1754, 'MDCCLIV'),
                     (1832, 'MDCCCXXXII'),
                     (1993, 'MCMXCIII'),
                     (2074, 'MMLXXIV'),
                     (2152, 'MMCLII'),
                     (2212, 'MMCCXII'),
                     (2343, 'MMCCCXLIII'),
                     (2499, 'MMCDXCIX'),
```