

O'REILLY®

Zrozumieć Bitcoin

Programowanie kryptowalut od podstaw



Helion 

Jimmy Song

Tytuł oryginału: Programming Bitcoin: Learn How to Program Bitcoin from Scratch

Tłumaczenie: Krzysztof Konatowicz

ISBN: 978-83-283-5923-9

© 2019 Helion SA

Authorized Polish translation of the English edition of Programming Bitcoin ISBN 9781492031499

© 2019 Jimmy Song

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Polish edition copyright © 2019 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/zrobit>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	11
Wprowadzenie	13
Rozdział 1. Ciąła skończone	23
Trochę matematyki wyższej	23
Definicja ciała skończonego	24
Definiowanie zbiorów skończonych	25
Tworzenie ciała skończonego w Pythonie	25
Ćwiczenie 1.	26
Arytmetyka modulo	26
Arytmetyka modulo w Pythonie	28
Dodawanie i odejmowanie w ciele skończonym	29
Ćwiczenie 2.	30
Programowanie dodawania i odejmowania w Pythonie	30
Ćwiczenie 3.	31
Mnożenie i potęgowanie w ciele skończonym	31
Ćwiczenie 4.	32
Ćwiczenie 5.	32
Programowanie mnożenia w Pythonie	32
Ćwiczenie 6.	33
Programowanie potęgowania w Pythonie	33
Ćwiczenie 7.	33
Dzielenie w ciele skończonym	33
Ćwiczenie 8.	35
Ćwiczenie 9.	35
Redefiniowanie potęgowania	36
Podsumowanie	37

Rozdział 2. Krzywe eliptyczne	39
Definicja	39
Kodowanie krzywych eliptycznych w Pythonie	44
Ćwiczenie 1.	45
Ćwiczenie 2.	45
Dodawanie punktów	45
Matematyka dodawania punktów	49
Programowanie dodawania punktów	51
Ćwiczenie 3.	52
Dodawanie punktów, gdy $x_1 \neq x_2$	52
Ćwiczenie 4.	54
Dodawanie punktów, gdy $x_1 \neq x_2$	54
Ćwiczenie 5.	54
Dodawanie punktów, gdy $P_1 = P_2$	54
Ćwiczenie 6.	55
Programowanie dodawania punktów, gdy $P_1 = P_2$	56
Ćwiczenie 7.	56
Programowanie jeszcze jednego przypadku	56
Podsumowanie	57
 Rozdział 3. Kryptografia krzywych eliptycznych	 59
Krzywe eliptyczne nad ciałem liczb rzeczywistych	59
Krzywe eliptyczne nad ciałami skończonymi	60
Ćwiczenie 1.	61
Programowanie krzywych eliptycznych nad ciałami skończonymi	62
Dodawanie punktów nad ciałami skończonymi	63
Programowanie dodawania punktów na krzywej nad ciałami skończonymi	64
Ćwiczenie 2.	64
Ćwiczenie 3.	65
Mnożenie skalarne dla krzywych eliptycznych	65
Ćwiczenie 4.	67
Mnożenie skalarne — druga odsłona	67
Grupy w matematyce	68
Element neutralny	68
Zamkniętość	69
Element odwrotny	70
Przemienność	70
Łączność	70
Ćwiczenie 5.	70
Programowanie mnożenia skalarnego	72

Definiowanie krzywej dla Bitcoina	74
Korzystanie z krzywej secp256k1	75
Kryptografia klucza publicznego	76
Podpisywanie i weryfikacja	77
Wpisywanie celu	78
Szczegóły weryfikacji	79
Weryfikacja podpisu	81
Ćwiczenie 6.	81
Programowanie weryfikacji podpisów	82
Szczegóły podpisywania	82
Tworzenie podpisu	83
Ćwiczenie 7.	84
Programowanie podpisywania komunikatów	84
Podsumowanie	86
Rozdział 4. Serializacja	87
Nieskompresowany format SEC	87
Ćwiczenie 1.	89
Skompresowany format SEC	89
Ćwiczenie 2.	92
Podpisy DER	92
Ćwiczenie 3.	94
Base58	94
Przesyłanie klucza publicznego	94
Ćwiczenie 4.	96
Format adresu	96
Ćwiczenie 5.	97
Format WIF	97
Ćwiczenie 6.	98
Porządek bajtowy (big- i little-endian) — dodatkowe informacje	98
Ćwiczenie 7.	98
Ćwiczenie 8.	98
Ćwiczenie 9.	99
Podsumowanie	99
Rozdział 5. Transakcje	101
Składniki transakcji	101
Wersja	103
Ćwiczenie 1.	104
Wejścia	104
Przetwarzanie pola ze skryptem	108
Ćwiczenie 2.	108

Wyjścia	108
Ćwiczenie 3.	110
Czas blokady	110
Ćwiczenie 4.	110
Ćwiczenie 5.	111
Kodowanie transakcji	111
Opłata transakcyjna	112
Obliczanie opłaty transakcyjnej	113
Ćwiczenie 6.	113
Podsumowanie	114
Rozdział 6. Język Script	115
Zasada działania języka Script	115
Jak działa Script?	116
Przykładowe operacje	117
Programowanie obsługi kodów operacji	118
Ćwiczenie 1.	118
Przetwarzanie pól ze skryptami	119
Programowanie analizatora składniowego i serializatora pól skryptów	120
Scalanie pól ze skryptami	121
Programowanie scalania skryptów	122
Skrypty standardowe	122
p2pk	123
Programowanie interpretera skryptów	125
Elementy stosu pod lupą	127
Ćwiczenie 2.	128
Problemy z p2pk	128
Rozwiązywanie problemów za pomocą p2pkh	130
p2pkh	130
Skrypty mogą być konstruowane dowolnie	134
Ćwiczenie 3.	136
Użyteczność skryptów	137
Ćwiczenie 4.	137
Wyzwanie: znalezienie kolizji SHA-1	137
Podsumowanie	137
Rozdział 7. Tworzenie i walidacja transakcji	139
Walidacja transakcji	139
Sprawdzanie wydania wejść	139
Sprawdzanie sumy wejść i sumy wyjść	140
Sprawdzanie podpisu	141

Ćwiczenie 1.	144
Ćwiczenie 2.	144
Weryfikacja całej transakcji	144
Tworzenie transakcji	145
Konstruowanie transakcji	145
Tworzenie transakcji	148
Podpisywanie transakcji	149
Ćwiczenie 3.	150
Tworzenie własnych transakcji w testnetcie	150
Ćwiczenie 4.	150
Ćwiczenie 5.	150
Podsumowanie	151
Rozdział 8. Pay-to-script-hash	153
Czysty multisig	153
Programowanie obsługi OP_CHECKMULTISIG	157
Ćwiczenie 1.	158
Problemy z czystym multisig	158
Pay-to-script-hash (p2sh)	158
Programowanie p2sh	164
Bardziej skomplikowane skrypty	165
Adresy	165
Ćwiczenie 2.	166
Ćwiczenie 3.	166
Weryfikacja podpisów p2sh	166
Ćwiczenie 4.	169
Ćwiczenie 5.	169
Podsumowanie	169
Rozdział 9. Bloki	171
Transakcje coinbase	171
Ćwiczenie 1.	172
ScriptSig	172
BIP0034	173
Ćwiczenie 2.	173
Nagłówki bloków	174
Ćwiczenie 3.	175
Ćwiczenie 4.	175
Ćwiczenie 5.	175
Wersja	175
Ćwiczenie 6.	176

Ćwiczenie 7.	176
Ćwiczenie 8.	176
Poprzedni blok	177
Korzeń drzewa skrótów	177
Znacznik czasu	177
Sekwencja bitowa	177
Wartość nonce	178
Dowód pracy	178
W jaki sposób górnik generuje nowe skróty?	179
Cel	179
Ćwiczenie 9.	180
Trudność	180
Ćwiczenie 10.	181
Sprawdzanie dowodu pracy	181
Ćwiczenie 11.	181
Zmiana trudności	181
Ćwiczenie 12.	183
Ćwiczenie 13.	183
Podsumowanie	183
Rozdział 10. Techniki sieciowe	185
Komunikaty sieciowe	185
Ćwiczenie 1.	186
Ćwiczenie 2.	186
Ćwiczenie 3.	187
Interpretowanie treści komunikatu	187
Ćwiczenie 4.	188
Uzgadnianie komunikacji w sieci	188
Łączenie się z siecią	188
Ćwiczenie 5.	191
Odbieranie nagłówków bloków	191
Ćwiczenie 6.	192
Odpowiedź z nagłówkami	192
Podsumowanie	194
Rozdział 11. Uproszczona weryfikacja płatności	195
Motywacja	195
Drzewo skrótów	196
Element nadrzędny	197
Ćwiczenie 1.	198
Poziom nadrzędny drzewa skrótów	198
Ćwiczenie 2.	199

Korzeń drzewa skrótów	199
Ćwiczenie 3.	200
Korzeń drzewa skrótów w blokach	200
Ćwiczenie 4.	201
Korzystanie z drzewa skrótów	201
Blok drzewa skrótów	202
Struktura drzewa skrótów	204
Ćwiczenie 5.	205
Programowanie obsługi drzewa skrótów	205
Komunikat sieciowy merkleblock	209
Ćwiczenie 6.	210
Wykorzystanie flag bitowych i skrótów	211
Ćwiczenie 7.	214
Podsumowanie	214
Rozdział 12. Filtry Blooma	215
Czym jest filtr Blooma?	215
Ćwiczenie 1.	217
Krok dalej	217
Filtry Blooma według BIP0037	218
Ćwiczenie 2.	220
Ćwiczenie 3.	220
Ładowanie filtra Blooma	220
Ćwiczenie 4.	220
Pobieranie bloków drzewa skrótów	221
Ćwiczenie 5.	221
Pobieranie interesujących nas transakcji	221
Ćwiczenie 6.	223
Podsumowanie	223
Rozdział 13. Segwit	225
Pay-to-witness-pubkey-hash (p2wphk)	225
Kowalność transakcji	225
Eliminowanie kowalności	226
Transakcje p2wphk	227
p2sh-p2wphk	230
Programowanie p2wphk i p2sh-p2wphk	234
Pay-to-witness-script-hash (p2wsh)	237
p2sh-p2wsh	241
Programowanie p2wsh i p2sh-p2wsh	246
Inne usprawnienia	247
Podsumowanie	248

Rozdział 14. Tematy zaawansowane i dalsze kroki	249
Proponowane tematy do dalszej nauki	249
Portfele	249
Kanały płatnicze i sieć Lightning	250
Społeczność	250
Proponowane dalsze projekty	251
Portfel testnetowy	251
Eksplorator bloków	251
Sklep internetowy	251
Biblioteka narzędzi	252
Poszukiwanie pracy	252
Podsumowanie	252
Dodatek A. Rozwiązania ćwiczeń	253
Skorowidz	287

Transakcje

Kwintesencją protokołu Bitcoin są transakcje. Transakcje to nic innego jak zapis przekazania jakiejś wartości pomiędzy kilkoma podmiotami. W rozdziale 6. przekonasz się, że owe podmioty to tak naprawdę inteligentne kontrakty (ang. *smart contracts*) — ale nie wszystko od razu! Zastanówmy się najpierw, czym są transakcje w protokole Bitcoin, jak wyglądają i jak będziemy je przetwarzać.

Składniki transakcji

Najogólniej rzecz biorąc możemy wyróżnić cztery podstawowe składniki każdej transakcji:

1. wersja,
2. wejścia,
3. wyjścia,
4. czas blokady.

Przyjrzyjmy się więc im bliżej. Wersja określa, jakie dodatkowe funkcje wykorzystuje transakcja, wejścia określają bitcoiny, które są przeznaczone do wydania, a wyjścia określają, dokąd wydawane bitcoiny mają trafić, natomiast czas blokady określa, od kiedy dana transakcja ma zacząć obowiązywać. Przeanalizujmy szczegółowo każdy z tych składników.

Rysunek 5.1 przedstawia szesnastkowy zapis typowej transakcji. Wyróżniono na nim poszczególne części transakcji.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Rysunek 5.1. Składniki transakcji: wersja, wejścia, wyjścia i czas blokady

Oznaczone różnymi kolorami fragmenty transakcji określają odpowiednio: wersję, wejścia, wyjścia i czas blokady.

Znając podstawową strukturę transakcji, możemy rozpocząć pisanie odpowiadającej jej klasy, którą nazwiemy Tx:

```
class Tx:
    def __init__(self, version, tx_ins, tx_outs, locktime, testnet=False):
        self.version = version
        self.tx_ins = tx_ins ❶
        self.tx_outs = tx_outs
        self.locktime = locktime
        self.testnet = testnet ❷
    def __repr__(self):
        tx_ins = ''
        for tx_in in self.tx_ins:
            tx_ins += tx_in.__repr__() + '\n'
        tx_outs = ''
        for tx_out in self.tx_outs:
            tx_outs += tx_out.__repr__() + '\n'
        return 'tx: {} \nversion: {} \ntx_ins: \n{} tx_outs: \n{} locktime: {}'.format(
            self.id(),
            self.version,
            tx_ins,
            tx_outs,
            self.locktime,
        )
    def id(self): ❸
        '''Czytelna dla człowieka heksadecymalna postać skrótu transakcji'''
        return self.hash().hex()
    def hash(self): ❹
        '''Binarny skrót w starej serializacji'''
        return hash256(self.serialize())[:-1]
```

- ❶ Wejścia i wyjścia są dość ogólnymi określeniami, dlatego określamy dokładniej ich rodzaj. Odpowiednie typy obiektów zdefiniujemy później.
- ❷ Aby móc w pełni sprawdzić poprawność transakcji, musimy jeszcze wiedzieć, w której sieci jest ona realizowana.
- ❸ Identyfikator id to numer identyfikujący transakcję, wykorzystywany na przykład przez eksploratory bloków. Jest to skrót hash256 transakcji w formacie szesnastkowym.
- ❹ Skrót generowany jest przez algorytm hash256 i jest serializowany w porządku bajtowym *little-endian*. Zwróć uwagę, że nie mamy jeszcze metody `serialize`, a więc dopóki jej nie napiszemy, kod nie będzie działał.

W dalszej części tego rozdziału będziemy zajmować się przetwarzaniem i analizowaniem transakcji. W tym momencie możemy więc już napisać podstawowy kod klasy:

```
class Tx:
    ...
    @classmethod ❶
    def parse(cls, serialization):
        version = serialization[0:4] ❷
    ...
```

- ❶ `parse` musi być metodą klasową, ponieważ serializacja będzie zwracać nową instancję obiektu Tx.
- ❷ Zakładamy, że zmienna `serialization` jest tablicą bajtową.

Coś takiego zapewne będzie działać, ale weźmy pod uwagę to, że transakcja może być bardzo duża. Lepiej byłoby więc, gdybyśmy mogli przetwarzać transakcję ze strumienia. Wtedy nie musielibyśmy odczytywać całej zserializowanej transakcji przed rozpoczęciem jej przetwarzania. Praca na strumieniu umożliwiłaby nam szybsze wykrywanie błędów i pozwoliłaby zwiększyć wydajność. Zatem kod przetwarzający transakcję będzie wyglądał mniej więcej tak:

```
class Tx:
    ...
    @classmethod
    def parse(cls, stream):
        serialized_version = stream.read(4) ❶
        ...
```

❶ Metoda `read` umożliwi nam przetwarzanie transakcji „w locie”, ponieważ nie będziemy musieli czekać na zakończenie operacji wejścia – wyjścia.

Ma to też zalety techniczne, gdyż strumień pozwoli nam przetwarzać dane pochodzące zarówno z gniazda sieciowego, jak i z pliku. Strumień możemy zacząć przetwarzać natychmiast i nie musimy czekać na przesłanie lub odczytanie wszystkich danych. Nasza metoda będzie obsługiwać wszystkie rodzaje strumieni i zwracać obiekt `Tx`.

Wersja

Przeznaczeniem numeru wersji (zobacz przykład na rysunku 5.2) jest przekazanie odbiorcy informacji o odmianie danej struktury lub elementu. Jeśli na przykład używasz systemu Windows 3.1, to jego numer wersji bardzo różni się od wersji systemu Windows 8 lub Windows 10. Zatem choć możesz opisać swój system samym słowem „Windows”, to podanie numeru wersji przekaże więcej informacji, w tym informacje o tym, jakie funkcje ten system obsługuje i za pomocą jakiego API można go programować.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Rysunek 5.2. Wersja

Na tej samej zasadzie transakcje Bitcoin mają też numery wersji. W przypadku protokołu Bitcoin numerem wersji transakcji jest zazwyczaj 1, choć w niektórych przypadkach może być to 2 (transakcje wykorzystujące kod operacji `OP_CHECKSEQUENCEVERIFY` z BIP0112 wymagają użycia wersji > 1).

Jak widzisz, numer wersji opisuje liczba szesnastkowa 01000000, której wartość nie jest równa 1. Ale jeśli zinterpretujemy ją w porządku *little-endian*, to okaże się, że liczba ta ma jednak wartość 1 (jeżeli nie pamiętasz dlaczego, zajrzyj do rozdziału 4.).

Ćwiczenie 1.

Napisz część zdefiniowanej wcześniej przez nas metody parse przetwarzającą numer wersji. Aby to zrobić poprawnie, musisz przekonwertować 4 bajty w porządku *little-endian* na liczbę całkowitą.

Wejścia

Każde wejście jest odwołaniem do wyjścia z jakiejś poprzedniej transakcji (rysunek 5.3). Wymaga to bliższego wyjaśnienia.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Rysunek 5.3. Wejścia

Wejściami transakcji są wyjścia z poprzednich transakcji. Zatem najpierw musimy otrzymać jakieś bitcoiny, aby móc je wydawać. Intuicyjnie ma to sens, bo nie możemy przecież wydawać pieniędzy, których wcześniej nie otrzymaliśmy. Wejścia opisują więc należące do nas bitcoiny. Każde wejście wymaga dwóch rzeczy:

- odwołania do bitcoinów, które wcześniej otrzymałeś;
- dowodu, że są one Twoje i że możesz je wydawać.

Do sprawdzenia drugiego warunku wykorzystywany jest algorytm ECDSA (zobacz rozdział 3.). Nie chcemy też, aby możliwe było sfalszowanie tych informacji, a więc większość wejść musi zawierać też podpisy, które mogą wygenerować wyłącznie posiadacze kluczy.

Pole wejść transakcji może zawierać więcej niż jedno wejście. Analogią może być tu użycie pojedynczego banknotu o nominale 100 zł do zapłaty za obiad w cenie 70 zł lub dwóch banknotów: 50 zł i 20 zł. W pierwszym przypadku potrzebne będzie nam tylko jedno wejście (jeden banknot), natomiast w drugim potrzebować będziemy dwóch. W niektórych sytuacjach transakcja może zawierać jeszcze więcej wejść. Wracając do naszego przykładu, moglibyśmy zapłacić za obiad w cenie 70 zł 14 monetami o wartości 5 zł albo rzucając sakiewkę z 7000 jednogroszówek. Odpowiadałoby to transakcji o 14 wejściach lub transakcji o 7000 wejściach.

Liczba wejść to kolejne pole transakcji (zobacz wyróżnienie na rysunku 5.4).

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Rysunek 5.4. Liczba wejść

Widzimy, że bajt ten ma wartość 01, co oznacza, że transakcja ma jedno wejście. Kuszące byłoby tu założenie, że wartość ta zajmuje zawsze 1 bajt, ale niestety tak nie jest. Jeden bajt ma 8 bitów, a więc nie dałoby się wyrazić nim liczby powyżej 255 wejść.

W tym momencie przyda się wiedza o wartościach *varint*. *Varint* to skrót od angielskiego wyrażenia *variable integer* (liczba całkowita o zmiennym formacie). Jest to metoda kodowania wartości całkowitej w postaci bajtowej w zakresie od 0 do $2^{64} - 1$. Moglibyśmy oczywiście na opis liczby wejść rezerwować zawsze 8 bajtów, ale byłoby to dużym marnotrawstwem, zwłaszcza że możemy spodziewać się, iż liczba wejść będzie stosunkowo mała (powiedzmy poniżej 200). Takie liczby wejść występują w większości normalnych transakcji, więc użycie *varint* da nam wymierną oszczędność. Sposób zapisu liczb w formacie *varint* przedstawiam w ramce.

Każde wejście zawiera cztery pola. Pierwsze dwa pola wskazują na wyjścia z poprzedniej transakcji, a dwa ostatnie definiują, w jaki sposób wyjścia z poprzednich transakcji mogą zostać wydane. Pola te to:

- ID poprzedniej transakcji,
- indeks poprzedniej transakcji,
- *ScriptSig*,
- kolejność.

Jak już wyjaśniłem, każde wejście zawiera odwołanie do wyjścia z poprzedniej transakcji. Identyfikatorem poprzedniej transakcji jest hash256 treści poprzedniej transakcji. Identyfikuje on jednoznacznie poprzednią transakcję, ponieważ prawdopodobieństwo kolizji tej funkcji skrótu jest niewyobrażalnie niskie.

Jak zobaczymy, każda transakcja musi mieć co najmniej jedno wyjście, choć może mieć ich wiele. Dlatego musimy dokładnie określić, które wyjścia z transakcji wydajemy, i to właśnie wskazuje wartość indeksu poprzedniej transakcji.

Zwróć uwagę, że identyfikator poprzedniej transakcji zajmuje 32 bajty oraz że indeks poprzedniej transakcji to 4 bajty. Obie wartości zakodowano w porządku *little-endian*.

Pole *ScriptSig* jest związane z językiem kontraktów inteligentnych Bitcoin — z językiem Script, który omówię dokładniej w rozdziale 6. Na razie *ScriptSig* traktuj jako zamkniętą skrytkę, którą otworzyć może tylko jego właściciel, czyli właściciel wyjścia transakcji. *ScriptSig* jest polem o zmiennej długości, co wyróżnia je na tle większości pól omówionych do tej pory. Pole o zmiennej długości wymaga, abyśmy dokładnie określili długość pola, dlatego jest ono poprzedzone wartością *varint* określającą właśnie jego długość.

Pole kolejności miało początkowo razem z polem czasu blokady (zobacz ramkę „Pola sequence i lock-time”) służyć do realizacji czegoś, co Satoshi nazwał transakcjami o wysokiej częstotliwości, ale obecnie jest wykorzystywane przez transakcje Replace-By-Fee (RBF) i kod operacji OP_CHECKSEQUENCEVERIFY. Wartość pola *sequence* jest również zapisywana w porządku *little-endian* i zajmuje 4 bajty.

Pola wejść zostały przedstawione na rysunku 5.5.

Liczby varint

Oto zasady zapisu liczb całkowitych w formacie *varint*:

- Jeśli liczba jest mniejsza niż 253, zapisujemy ją jako 1 bajt (np. 100 → 0x64).
- Jeśli liczba mieści się w przedziale od 253 do $2^{16} - 1$, zapisujemy bajt o wartości 253 (fd), a następnie liczbę w 2 bajtach w porządku *little-endian* (np. 255 → 0xfdf00, 555 → 0xfd2b02).
- Jeśli liczba mieści się w przedziale od 2^{16} do $2^{32} - 1$, zapisujemy bajt o wartości 254 (fe), a następnie zapisujemy liczbę w 4 bajtach w porządku *little-endian* (np. 70015 → 0xfe7f110100).
- Jeśli liczba mieści się w przedziale od 2^{32} do $2^{64} - 1$, zapisujemy bajt o wartości 255 (ff), a następnie zapisujemy liczbę w 8 bajtach w porządku *little-endian* (np. 18005558675309 → 0xff6dc7ed ↪ 3e60100000).

Do przetwarzania i serializacji wartości *varint* wykorzystamy dwie funkcje z pliku *helper.py*:

```
def read_varint(s):
    """read_varint odczytuje wartość typu varint ze strumienia"""
    i = s.read(1)[0]
    if i == 0xfd:
        # 0xfd oznacza, że 2 kolejne bajty określają wartość
        return little_endian_to_int(s.read(2))
    elif i == 0xfe:
        # 0xfe oznacza, że 4 kolejne bajty określają wartość
        return little_endian_to_int(s.read(4))
    elif i == 0xff:
        # 0xff oznacza, że 8 kolejnych bajtów określa wartość
        return little_endian_to_int(s.read(8))
    else:
        # Każda inna wartość to po prostu liczba całkowita
        return i

def encode_varint(i):
    """Koduje wartość integer, w formacie varint"""
    if i < 0xfd:
        return bytes([i])
    elif i < 0x10000:
        return b'\xfd' + int_to_little_endian(i, 2)
    elif i < 0x100000000:
        return b'\xfe' + int_to_little_endian(i, 4)
    elif i < 0x10000000000000000:
        return b'\xff' + int_to_little_endian(i, 8)
    else:
        raise ValueError('za duża wartość typu int: {}'.format(i))
```

Funkcja `read_varint` odczytuje ze strumienia i zwraca liczbę całkowitą. Funkcja `encode_varint` wykonuje odwrotną operację, czyli dla podanej liczby całkowitej zwraca ją w formacie *varint*.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Rysunek 5.5. Pola wejść: identyfikator poprzedniej transakcji, indeks poprzedniej transakcji, ScriptSig i pole kolejności

Pola sequence i locktime

Początkowo Satoshi chciał, aby pola *sequence* (kolejność) i *locktime* (czas blokady) były wykorzystywane do obsługi tak zwanych transakcji o wysokiej częstotliwości (ang. *high-frequency trades*). Satoshi wyobrażał sobie, że będą one sposobem rozliczania transakcji dwukierunkowych realizowanych pomiędzy stronami, bez konieczności zapisywania wielu transakcji w łańcuchu bloków. Na przykład jeśli Alicja płaci za coś Robertowi x bitcoinów, a później Robert płaci Alicji y bitcoinów za coś innego (powiedzmy, że $x > y$), to zamiast zapisywania oddzielnych transakcji w łańcuchu Alicja może po prostu zapłacić Robertowi $x - y$. To samo moglibyśmy zrobić, gdyby Alicja i Robert mieli do rozliczenia pomiędzy sobą 100 transakcji. W ten sposób moglibyśmy skompresować wiele transakcji do jednej.

O to właśnie chodziło Satoshiemu, czyli o aktualizowaną ciągle miniksięgę rejestrującą transakcje pomiędzy dwiema zaangażowanymi w jakiś obrót stronami, która zostanie później rozliczona w łańcuchu bloków. Zamiarem Satoshiego było użycie pól *sequence* i *locktime* do aktualizowania transakcji o wysokiej częstotliwości za każdym razem, gdy dochodziło do kolejnej płatności pomiędzy dwiema stronami. Transakcja taka miałaby dwa wejścia (jedno od Alicji i jedno od Roberta) i dwa wyjścia (jedno do Alicji i jedno do Roberta). Rozpoczynałaby się od wartości pola *sequence* 0 i dużego czasu blokady (powiedzmy 500 bloków od teraz, a więc obowiązywałaby za 500 bloków). Byłaby to transakcja bazowa, w której Alicja i Robert otrzymują te same kwoty, które zadysponowali.

Po pierwszej transakcji, w której Alicja płaci Robertowi x bitcoinów, pole *sequence* każdego wejścia miałoby wartość 1. Po drugiej transakcji, w której Robert płaci Alicji y bitcoinów, pole *sequence* każdego wejścia miałoby wartość 2. W ten sposób moglibyśmy umieszczać wiele płatności w pojedynczej transakcji zapisywanej w łańcuchu bloków, pod warunkiem że zostałyby one zrealizowane przed upływem czasu blokady.

Niestety, chociaż to dość sprytny pomysł, okazuje się, że zły górnik mógłby dość łatwo nadużyć takiej możliwości. Powiedzmy, że w naszym przykładzie Robert jest złym górnikiem. Mógłby zignorować zaktualizowaną transakcję o wartości pola *sequence* 2 i wykopać transakcję z wartością pola *sequence* 1, oszukując Alicję na y bitcoinów.

Znacznie lepsze rozwiązanie powstało później. Chodzi tu o „kanały płatności”, które są podstawą sieci Lightning.

Teraz, gdy wiesz już, jakie pola wchodzą w skład transakcji, możemy rozpocząć programowanie klasy TxIn w Pythonie:

```
class TxIn:
    def __init__(self, prev_tx, prev_index, script_sig=None, sequence=0xffffffff):
        self.prev_tx = prev_tx
        self.prev_index = prev_index
        if script_sig is None: ❶
            self.script_sig = Script()
        else:
            self.script_sig = script_sig
        self.sequence = sequence
    def __repr__(self):
        return '{}:{}'.format(
            self.prev_tx.hex(),
            self.prev_index,
        )
```

❶ Domyślnie używamy pustego pola *ScriptSig*.

Mamy tu kilka rzeczy, na które warto zwrócić uwagę. Przede wszystkim kwota żadnego wejścia nie została określona. Nie mamy pojęcia, jaka liczba bitcoinów jest wydawana, dopóki nie sprawdzimy w łańcuchu bloków transakcji, z których bitcoiny będziemy wydawać. Co więcej, nie wiedząc nic o poprzedniej transakcji, nie wiemy nawet, czy transakcja ta otwiera „odpowiednią skrytkę”. Każdy węzeł musi weryfikować, czy ta transakcja otwiera odpowiednią skrytkę i czy ktoś nie próbuje w niej wydać bitcoinów, które nie istnieją. Jak to zrobić, opiszę w rozdziale 7.

Przetwarzanie pola ze skryptem

Sposób przetwarzania języka Script opiszę bardziej szczegółowo w rozdziale 6. Na razie wystarczy Ci wiedza, jak w Pythonie można uzyskać obiekt Script z ciągu szesnastkowego:

```
>>> from io import BytesIO
>>> from script import Script ❶
>>> script_hex = ('6b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b02774
↳ 57c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631e
↳ 3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a')
>>> stream = BytesIO(bytes.fromhex(script_hex))
>>> script_sig = Script.parse(stream)
>>> print(script_sig) 3045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277
↳ 457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed010349fc4e631
↳ e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a
```

- ❶ Klasę Script omówię dokładniej w rozdziale 6. Na razie wystarczy wiedzieć, że metoda Script.
↳ parse utworzy obiekt, którego potrzebujemy.

Ćwiczenie 2.

Napisz część metody parse przetwarzającą wejścia w Tx i metodę parse dla TxIn.

Wyjścia

Jak już wspomniałem w poprzednim punkcie, wyjścia określają, dokąd mają trafić bitcoiny w wyniku transakcji. Każda transakcja musi mieć co najmniej jedno wyjście. Ale po co komuś więcej niż jedno wyjście? Giełdy kryptowalut mogą na przykład łączyć transakcje w pule i wypłacać środki wielu osobom jednocześnie zamiast generować pojedyncze transakcje dla każdej osoby wypłacającej z giełdy swoje bitcoiny.

Podobnie jak w przypadku wejść, serializacja wyjść rozpoczyna się od zapisu wartości w formacie *varint* określającej liczbę wyjść (rysunek 5.6).

```
0100000001813f79011ac80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Rysunek 5.6. Liczba wyjść

Zbiór UTXO

Nazwa UTXO pochodzi od angielskiego wyrażenia *unspent transaction output* (niewydane wyjścia transakcyjne). Zbiór UTXO jest kompletnym zbiorem wszystkich niewydanych wyjść transakcyjnych w danym momencie. Powodem, dla którego zbiór UTXO jest ważny, jest to, że opisuje on wszystkie bitcoiny, które mogą zostać wydane w danej chwili. Innymi słowy opisuje bitcoiny, które znajdują się aktualnie w obiegu. Pełne węzły w sieci Bitcoin muszą monitorować zbiór UTXO. Indeksowanie zbioru UTXO znacznie przyspiesza weryfikację nowych transakcji.

Na przykład można łatwo egzekwować zakaz podwójnego wydatkowania poprzez sprawdzenie wyjścia poprzedniej transakcji w zbiorze UTXO. Jeśli wejściem nowej transakcji jest wyjście transakcji, której nie ma w zbiorze UTXO, oznacza to próbę podwójnego wydania lub nieprawidłowe wyjście, a zatem transakcja taka jest nieprawidłowa. Przechowywanie zbioru UTXO pod ręką również przydaje się do weryfikowania poprawności transakcji. Jak zobaczysz w rozdziale 6., aby zweryfikować transakcję, musimy sprawdzić kwotę i wartość pola *ScriptPubKey* wyjścia poprzedniej transakcji, a więc posiadanie kopii danych UTXO może przyspieszyć walidację transakcji.

Każde wyjście ma dwa pola: *amount* i *ScriptPubKey*. Wartość pola *amount* (kwota) to dokładnie liczba bitcoinów, które są przekazywane w danej transakcji. Jest ona wyrażana w jednostce satoshi (1 satoshi to 1/100 000 000 bitcoina). Umożliwia to bardzo precyzyjne dzielenie bitcoinów (z dokładnością do 0,03 grosza, według wartości bitcoina w połowie 2019 r.). Bezwzględne maksimum dla tej kwoty to górny limit, czyli maksymalna liczba 21 milionów bitcoinów wyrażona w satoshi, czyli 2 100 000 000 000 000 (2100 bilionów) satoshi. Liczba ta jest większa niż 2^{32} (około 4,3 miliarda) i dlatego jest przechowywana w 64 bitach, czyli w 8 bajtach. Kwota ta jest serializowana w porządku *little-endian*.

Pola *ScriptPubKey*, podobnie jak *ScriptSig*, związane są z językiem z inteligentnych kontraktów Bitcoin — językiem Script. Pole *ScriptPubKey* również należy traktować jako zamkniętą skrytkę, którą może otworzyć tylko posiadacz klucza. To jakby jednokierunkowy sejf, w którym każdy coś może zdeponować, ale otworzyć go może tylko właściciel. Przyjrzymy się temu dokładniej w rozdziale 6. Podobnie jak *ScriptSig*, także *ScriptPubKey* jest polem o zmiennej długości i jest poprzedzone długością pola zapisaną w formacie *varint*.

Całe wyjście wygląda jak na rysunku 5.7.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Rysunek 5.7. Całe pole wyjścia zawierające kwotę i pole *ScriptPubKey* — to wyjście ma indeks 0

Możemy już rozpocząć pisanie klasy *TxOut*:

```
class TxOut:
    def __init__(self, amount, script_pubkey):
        self.amount = amount
```

```

self.script_pubkey = script_pubkey
def __repr__(self):
    return '{}:{}'.format(self.amount, self.script_pubkey)

```

Ćwiczenie 3.

Napisz część metody `parse` przetwarzającą wyjścia w klasie `Tx` oraz metodę `parse` dla `TxOut`.

Czas blokady

Pole *locktime* (czas blokady) umożliwia opóźnienie transakcji. Transakcja z wartością 600 000 w tym polu nie będzie mogła trafić do łańcucha bloków aż do bloku 600 001. Pierwotnym zastosowaniem tego pola miała być obsługa transakcji o wysokiej częstotliwości (zobacz ramkę „Pola sequence i locktime”), jednak transakcje te okazały się niebezpieczne. Jeśli czas blokady jest większy lub równy 500 000 000, oznacza to, że jest on wyrażony uniksowym znacznikiem czasu. Jeśli jest mniejszy od 500 000 000, to jest numerem bloku. Dzięki temu transakcje można podpisać, ale nie będzie można ich spżytkować aż do pewnego momentu wyrażonego albo czasem uniksowym, albo numerem bloku.



Kiedy pole *locktime* jest ignorowane?

Pole *locktime* jest ignorowane, jeśli wartości pól *sequence* każdego wejścia to `ffffffff`.

Pole to jest serializowane w porządku *little-endian* w 4 bajtach (rysunek 5.8).

```

0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000
00006b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457
c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed012103
49fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffffff02a135
ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000
001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600

```

Rysunek 5.8. Pole *locktime* (czas blokady)

Głównym problemem związanym z używaniem pola *locktime* jest to, że odbiorca transakcji nie ma pewności, czy transakcja będzie poprawna po upływie określonego przez to pole czasu. Jest to podobne do czeku bankowego wystawionego z przyszlą datą, który może nie zostać uznany. Posiadacz może wydać swoje bitcoiny wskazane jako wejścia transakcji, zanim transakcja z ustawionym czasem blokady trafi do łańcucha bloków, co spowoduje jej unieważnienie po upływie czasu blokady.

Przed BIP0065 zastosowania tego pola były ograniczone. W BIP0065 wprowadzono kod operacji `OP_CHECKLOCKTIMEVERIFY` zwiększający przydatność pola *locktime* dzięki uniemożliwieniu wydania wyjść przez określony czas blokady.

Ćwiczenie 4.

Napisz część metody `parse` klasy `Tx`, która przetwarza pole *locktime*.

Ćwiczenie 5.

Podaj *ScriptSig* drugiego wejścia, *ScriptPubKey* pierwszego wyjścia i kwotę drugiego wyjścia dla tej transakcji:

```
010000000456919960ac691763b88d3d3bcea9ad6ecaf875df5339e148a1fc61c6ed7a069e0100000006a4730440220
4585bcdef85e6b1c6af5c2669d4830f86e42dd205c0e089bc2a821657e951c002201024a10366077f87d6bce1f710
0ad8cfa8a064b39d4e8fe4ea13a7b71aa8180f012102f0da57e85eec2934a82a585ea337ce2f4998b50ae699dd79f5
880e253dafafb7fefffffebf8f51f4038dc17e6313cf831d4f02281c2a468bde0fafd37f1bf882729e7fd300000000
6a47304402207899531a52d59a6de200179928ca900254a36b8dff8bb75f5f5d71b1cdc26125022008b422690b8461
cb52c3cc30330b23d574351872b7c361e9aae3649071c1a7160121035d5c93d9ac96881f19ba1f686f15f009ded7c6
2efe85a872e6a19b43c15a2937fefffff567bf40595119d1bb8a3037c356efd56170b64cbcc160fb028fa10704b45
d77500000006a47304402204c7c7818424c7f7911da6cddc59655a70af1cb5eaf17c69dadbfcc74ffa0b662f022075
99e08bc8023693ad4e9527dc42c34210f7a7d1d1ddfc8492b654a11e7620a0012102158b46fbdff65d0172b7989aec
8850aa0dae49abfb84c81ae65b251a58ace5cfefffffd63a5e6c16e620f86f375925b21caba7f736c779f88fd04dc
ad51d26690f7f345010000006a47304402200633ea0d3314bea0d95b3cd8dadb2ef79ea8331ffe1e61f762c0f6daea
0fabde022029f23b3e9c30f080446150b23852028751635dcee2be669c2a1686a4b5edf304012103ffd6f4a67e94ab
a353a00882e563ff2722eb4cff0ad6006e86ee20dfe7520d55fefffff0251430f0000000001976a914ab0c0b2e98
b1ab6dbf67d4750ba0a56244948a87988ac005a620200000001976a9143c82d7df364eb6c75be8c80df2b3eda8db57
397088ac46430600
```

Kodowanie transakcji

Umiesz już przetwarzać i interpretować zapis transakcji. Teraz chcielibyśmy więc napisać kod realizujący operację odwrotną, czyli serializację transakcji. Zacznijmy od klasy TxOut:

```
class TxOut:
    ...
    def serialize(self): ❶
        """Zwraca bajtówą serializację wyjścia transakcji"""
        result = int_to_little_endian(self.amount, 8)
        result += self.script_pubkey.serialize()
        return result
```

❶ Zamierzamy wykonać serializację obiektu TxOut, zapisując go jako ciąg bajtów.

Następnie możemy przejść do TxIn:

```
class TxIn:
    ...
    def serialize(self):
        """Zwraca bajtówą serializację wejścia transakcji"""
        result = self.prev_tx[::-1]
        result += int_to_little_endian(self.prev_index, 4)
        result += self.script_sig.serialize()
        result += int_to_little_endian(self.sequence, 4)
        return result
```

Na koniec możemy zaprogramować serializację całej transakcji Tx:

```
class Tx:
    ...
    def serialize(self):
        """Zwraca bajtówą serializację transakcji"""
        result = int_to_little_endian(self.version, 4)
        result += encode_varint(len(self.tx_ins))
        for tx_in in self.tx_ins:
```

```

        result += tx_in.serialize()
    result += encode_varint(len(self.tx_outs))
    for tx_out in self.tx_outs:
        result += tx_out.serialize()
    result += int_to_little_endian(self.locktime, 4)
    return result

```

Do serializacji Tx użyliśmy metod `serialize` z obu klas, `TxIn` i `TxOut`.

Zauważ, że nigdzie nie podano opłaty transakcyjnej. Jest tak, ponieważ opłata ta jest obliczana. Sposób jej obliczania przedstawię w następnym punkcie.

Opłata transakcyjna

Jedną z zasad konsensusu protokołu Bitcoin jest to, że dla każdej transakcji niebędącej transakcją *coinbase* (więcej na temat transakcji *coinbase* przeczytasz w rozdziale 9.) suma wejść musi być większa lub równa sumie wyjść. Dlaczego po prostu nie wymusić równości tych sum? Dlatego, że gdyby każda transakcja miała zerowy koszt, nie byłoby żadnej motywacji dla górników, aby dołączali transakcje do bloków (zobacz rozdział 9.). Opłaty są więc sposobem motywowania górników do włączania transakcji do bloków. Transakcje spoza bloków (tak zwane *transakcje mempool*) nie należą do łańcucha bloków i nie są ostateczne.

Opłata transakcyjna jest po prostu sumą wejść pomniejszoną o sumę wyjść. Różnicę między tymi kwotami zatrzymuje sobie górnik. Ponieważ wejścia nie mają pola kwoty, ich wartości musimy odszukać. Wymaga to dostępu do łańcucha bloków, a konkretnie do zbioru UTXO. Jeśli Twój węzeł nie jest węzłem pełnym, to uzyskanie tych danych może być trudne, musisz bowiem zaufać innemu podmiotowi, który udostępni Ci takie informacje.

Utworzymy nową klasę, która obsłuży to zadanie — nazwiemy ją `TxFetcher`:

```

class TxFetcher:
    cache = {}
    @classmethod
    def get_url(cls, testnet=False):
        if testnet:
            return 'http://testnet.programmingbitcoin.com'
        else:
            return 'http://mainnet.programmingbitcoin.com'
    @classmethod
    def fetch(cls, tx_id, testnet=False, fresh=False):
        if fresh or (tx_id not in cls.cache):
            url = '{}{}/tx/{}/.hex'.format(cls.get_url(testnet), tx_id)
            response = requests.get(url)
            try:
                raw = bytes.fromhex(response.text.strip())
            except ValueError:
                raise ValueError('nieoczekiwana odpowiedź: {}'.format(response.text))
            if raw[4] == 0:
                raw = raw[:4] + raw[6:]
                tx = Tx.parse(BytesIO(raw), testnet=testnet)
                tx.locktime = little_endian_to_int(raw[-4:])
            else:
                tx = Tx.parse(BytesIO(raw), testnet=testnet)

```

```

if tx.id() != tx_id: ❶
    raise ValueError('różne identyfikatory : {} != {}'.format(tx.id(),
                                                              tx_id))
cls.cache[tx_id] = tx
cls.cache[tx_id].testnet = testnet
return cls.cache[tx_id]

```

❶ Sprawdzamy, czy identyfikator jest tym, czego oczekujemy.

Być może się zastanawiasz, dlaczego zamiast pobrać samo wyjście dla danej transakcji pobieramy całą transakcję. Wynika to stąd, że nie chcemy uzależnić swojego bezpieczeństwa od zaufania do jakiegokolwiek trzeciej strony! Pobierając całą transakcję, możemy sami zweryfikować identyfikator transakcji (hash256 jej treści), co da nam pewność, że faktycznie otrzymaliśmy tę transakcję, o którą nam chodziło. Nie byłoby to możliwe, gdybyśmy nie pobrali całej transakcji.



Dlaczego minimalizujemy zaufanie do stron trzecich?

Jak elokwentnie ujął to Nick Szabo w swoim artykule *Trusted Third Parties are Security Holes* (<https://nakamotoinstitute.org/trust-third-parties/>), zaufanie stronie trzeciej i uznanie, że dostarcza ona poprawne dane, *nie* jest dobrą praktyką z punktu widzenia bezpieczeństwa. Choć taka trzecia strona może początkowo zachowywać się właściwie, to nigdy nie wiemy, czy kiedyś nie zostanie zhakowana, czy któryś z jej pracowników nie zacznie działać przeciwko niej lub czy nie zacznie ona wdrażać zasad niezgodnych z naszymi interesami. Tym, co sprawia, że protokół Bitcoin jest bezpieczny, *nie* jest zaufanie, lecz weryfikacja danych, które otrzymujemy.

Możemy teraz napisać odpowiednią metodę w klasie TxIn pobierającą poprzednią transakcję oraz metody pobierające kwoty z wyjść poprzedniej transakcji i wartość pola *ScriptPubKey* (które później wykorzystamy w rozdziale 6.):

```

class TxIn:
    ...
    def fetch_tx(self, testnet=False):
        return TxFetcher.fetch(self.prev_tx.hex(), testnet=testnet)
    def value(self, testnet=False):
        """Pobiera wartość wyjścia, sprawdzając skrót transakcji. Zwraca kwotę w jednostkach satoshi"""
        tx = self.fetch_tx(testnet=testnet)
        return tx.tx_outs[self.prev_index].amount
    def script_pubkey(self, testnet=False):
        """Pobiera pole ScriptPubKey, sprawdzając skrót transakcji. Zwraca obiekt Script"""
        tx = self.fetch_tx(testnet=testnet)
        return tx.tx_outs[self.prev_index].script_pubkey

```

Obliczanie opłaty transakcyjnej

Teraz, gdy mamy już w klasie TxIn metodę *value*, która pozwala uzyskać dostęp do informacji o tym, ile bitcoinów jest w poszczególnych wejściach transakcji, możemy obliczyć opłatę za transakcję.

Ćwiczenie 6.

Napisz metodę *fee* dla klasy Tx.

Podsumowanie

Wiesz już, jak przetwarzać, interpretować oraz serializować transakcje. Zdefiniowaliśmy też znaczenie poszczególnych pól transakcji. Dwa pola nadal wymagają bliższych wyjaśnień, oba związane są z językiem kontraktów inteligentnych protokołu Bitcoin, czyli z językiem Script. Tym tematem zajmiemy się właśnie w rozdziale 6.

%, 28
||, 197
1 satoshi, 109

A

adres, 165
 IP, 187
algorytm podpisu, 78
Ammous Saifedean, 21
amount, 109
Andresen Gavin, 173
AntMiner S9, 179
anyone-can-spend, 233
Armory, 249
arytmetyka modulo, 26, 31
ASICBOOST, 179
asymetria, 68
atak urodzinowy, 81

B

Base58, 95
Bcoin, 250
Bech32, 95, 230
biblioteka narzędzi, 252
big-endian, 87, 88, 98
BIP0009, 175, 176
BIP001, 164
BIP0016, 159, 163
BIP0034, 172, 173, 175
BIP0037, 218
BIP0065, 110
BIP0066, 175
BIP0091, 176
BIP0141, 176, 225

BIP0143, 144, 225, 236
BIP0173, 95, 230
Bishop Bryan, 21
Bitcoin, 20
 Core 0.3.11, 141
Bitcoin Core, 250
Bitcoin Meetup, 20
BitcoinJ, 251
BitcoinJS, 251
blockchain, 177
blok, 174
 drzewa skrótów, 202
 filtrowany, 221
 genesis, 177
 początkowy, 177
bloki, 171
BOLT, 250
Bowman Casey, 21
brain wallet, 83
Braunberger Thomas, 21
btcd, 250
BTCPay, 251

C

C#, 251
C++, 250
Calvin Jim, 21
Cascarilla Chad, 20
Caswell Aaron, 20
cel, 179, 180
Chen Albert, 21
ciało, 253
 skończone, 29, 59, 62, 63
 zamknięte, 29
ciało skończone, 24, 25
Cobain Eduardo, 21

coefficient, 179
coinbase, 112, 140, 171
Coinomi, 250
Cole Napoleon, 20, 21
Cole Tipton, 21
Cole Will, 21
Corallo Matt, 21
CVE-2010-5139, 141
CVE-2012-2459, 197
cykliczna grupa skończona, 66
czas blokady, 101, 107, 110

D

Daftuar Suhas, 21
Demeester Tuur, 21
denial-of-service, 197
DER, 92, 94
difficulty, 180
 adjustment period, 181
Dilley Johnny, 21
Distinguished Encoding Rules, 92
dodawanie punktów, 45, 48
double SHA-1, 81
dowód
 pracy, 178
 włączenia, 195
drzewo skrótów, 195, 196

E

ECDSA, 78, 87, 104, 147
Edge, 250
eksplorator bloków, 251
Electrum, 250
element
 ciała, 25
 neutralny, 49
 odwrotny, 49
Elliptic Curve Digital Signature Algorithm, 78
envelope, 189
epoka, 176
exponent, 179

F

Falke Marco, 21
fba4c795, 219
FieldElement, 25, 253

filtered block, 221
filtry Blooma, 187, 215
flaga
 bitowa, 211
 przekazywania, 220
Flaxman Michael, 20, 21
Flowers Jeff, 21
format importu portfela, 97
funkcja skrótu, 197

G

genesis, 177
Go, 250
Goldstein Michael, 20
górn timer, 112, 147, 179
grupa, 66
 skończona, 66

H

Hanson Spencer, 21
Harding David, 21
hash, 79
hash160, 160
hash256, 81, 102
headers, 192
hierarchical deterministic, 249
high-frequency trades, 107
Hunt Thomas, 21

I

ID poprzedniej transakcji, 105
identyfikacja sieci, 185
incydent przepełnienia, 141
indeks poprzedniej transakcji, 105
integer, 84
inteligentne kontrakty, 101
IPv4, 187
IPv6, 187

J

Java, 251
JavaScript, 250, 251
Jupyter Notebook, 16

K

kanały płatnicze, 250
Karpeles Mark, 225
Kiss Richard, 20
klient referencyjny, 250
klucz
 prywatny, 77, 84, 97
 publiczny, 77, 83, 84, 147
kod źródłowy, 15
kolejność, 105, 107
kolizja SHA-1, 269
kompletność w sensie Turinga, 164
komunikat, 187
 sieciowy, 209
konkatenacja, 197
kontrakt, 79
 inteligentny, 225
koparki kryptowalut, 179
koperta, 189
korzeń drzewa skrótów, 177, 195, 197, 199
kowalność, 84
 transakcji, 157, 225
Krawisz Daniel, 20
krzywa
 ciągła, 41
 eliptyczna, 39, 40, 42, 62, 74
 sześcienna, 41
kwadratowa złożoność obliczeniowa, 247
kwota, 109, 140

L

Lau Johnson, 21
Left, 197
Les Jason, 21
Lewis Parker, 21
Libbittcoin, 250
liczba
 całkowita
 bez znaku, 141
 o zmiennym formacie, 105
 ze znakiem, 141
 elementów, 221
 pierwsza, 32
liczby rzeczywiste, 59
Lightning, 107, 250
Linia prosta, 63
Linux, 15
Liotti Brian, 21

liście, 197
Litecoin, 185
little-endian, 88, 98, 102
Liu Ken, 12, 20
locktime, 107, 110, 265

L

łańcuch bloków, 177
łączność, 50

M

macOS, 15
mainnet, 96, 145
malleability, 84, 225
małe twierdzenie Fermata, 34, 81
Marco Falke, 21
Maxwell Greg, 21
median time past, 177
mempool, 112, 147
merkle
 block, 202
 tree, 196
 root, 177
merkleblock komunikat, 209
miękki fork, 225
Mizrahi Alex, 20
mnożenie skalarne, 65, 66, 67, 72
modulo, 26, 31
moneta testnetowa, 99
Moon Justin, 21
Morcos Alex, 21
MTP, 177
multisig, 141, 153, 155, 237
 czysty, 158
murmur3, 219

N

nachylenie
 linii, 53
 stycznej, 55
nagłówki, 192
 bloku, 174
nagroda za blok, 171
Nair Raj, 20
Nakamoto Satoshi, 20
nested Segwit, 230
network magic, 185

Newbery John, 21
niewydane wyjścia transakcyjne, 109
nonce, 178, 179
nowe bitcoiny, 139
number used only once, 178

O

odwrotność
 addytywna, 24
 multiplikatywna, 24
okres dopasowania trudności, 181
OnionCat, 187
OP_CHECKMULTISIG, 157
opłata, 147
 transakcyjna, 112
 ujemna, 140
 za transakcję, 113
opóźnienie transakcji, 110

P

p2pk, 147
p2pkh, 95, 147, 154
p2sh, 158, 160, 164, 230
p2sh-p2wsh, 241
p2wpkh, 225, 230
p2wsh, 237
Parent, 197
pay-to-pubkey-hash, 95
pay-to-script-hash, 158, 161
pay-to-witness-pubkey-hash, 225
pay-to-witness-script-hash, 237
peer-to-peer, 185
Piscitello Alan, 21
PlayStation 3, 85
podpis, 83
 Schnorra, 153
podwójne wydatkowanie, 139
Poelstra Andrew, 21
Point, 44, 62
pole korzenia drzewa skrótów, 195
poprawność podpisu, 272
portfel, 249, 251
 deterministyczne, 249
 hierarchiczno-deterministyczne, 249
 mnemotechniczny, 83
portfolio, 252
poziom nadrzędny, 197
 drzewa skrótów, 198

praca
 lokalna, 252
 zdalna, 252
problem
 kwadratowej złożoności funkcji skrótu, 144
 logarytmu dyskretnego, 66
 podwójnego wydatkowania, 171
protokół, 185
przechodzenie drzewa, 206
przemienność, 50
przepełnienie zmiennej, 141
pubkey, 83
pule transakcji, 147
punkt, 44
 w nieskończoności, 49
pycoin, 250
Python, 13, 15, 250

Q

quadratic hashing problem, 144

R

random, 78
RBF, 105
RedeemScript, 159, 160, 230
reguły zgodności, 145
Reiner Alan, 20
relay, 220
Replace-By-Fee, 105
RFC 6979, 85
Right, 197
ripemd160, 96, 147
rozwiniecie binarne, 73
równanie
 kwadratowe, 39
 sześciennne, 39
rząd zbioru, 24

S

ScriptPubKey, 109, 113, 142, 215
ScriptSig, 105, 139, 142
SEC, 87, 94
secp256k1, 44, 60, 74
seed, 219, 249
segregated witness, 225
Segwit, 95, 147, 176, 225

- sequence, 107
- serializacja, 87
- serialize, 102
- SHA-1, 81
- sha256, 147
- sieć, 185
 - główna, 96
 - testowa, 96
- Bitcoin, 145
- signature hash, 79
- signed integer, 141
- Silberstein Eric, 20
- skalar, 65
- sklep, 251
- skompresowany format SEC, 90
- skrót, 79
 - lewy, 197
 - nadrzędny, 197
 - podpisu, 79
 - prawy, 197
- skrypt wypłaty, 159
- skrypty z wielopodpisem, 237
- smart contracts, 101
- Song Jimmy, 12
- SPV, 214
- Standards for Efficient Cryptography, 87
- styczna, 54, 56
- suma kontrolna, 96
- Szabo Nick, 113

S

- środowisko, 15

T

- target, 180
- tBTC, 145
- testnet, 96, 145, 251
 - bitcoin faucet, 150
- Texas Bitcoin, 20
- The Value Overflow Incident, 141
- Tone Vays, 21
- tożsamość
 - addytywna, 24
 - multiplikatywna, 24
- transaction malleability, 225
- transakcje, 101, 105, 148
 - coinbase, 171
 - mempool, 112

- o wysokiej częstotliwości, 107
- spoza bloków, 112
- zależne, 226

- Trezor, 250
- trudność, 180
- tweak, 218
- typ elementu, 221

U

- unsigned integer, 141
- unspent transaction output, 109
- uproszczona weryfikacja płatności, 214
- UTXO, 109, 139, 270

V

- ValueError, 26
- van der Laan Wladimir, 21
- variable integer, 105
- varint, 105, 108, 193
- version field rolling, 179

W

- walidacja
 - podpisu, 168
 - transakcji, 139
- Wallet Import Format, 97
- wartość losowa, 78
- wejścia, 101, 104
- wersja, 101, 103, 175
- weryfikacja podpisów, 166
- węzeł, 185, 211
- węzeł wewnętrzny, 197
- wielopodpis, 153
- WIF, 97, 263
- Windows, 15
- witness, 226
- Wuille Pieter, 21
- wydaje, kto chce, 233
- wyjścia, 101, 108

Z

- zagnieżdżony Segwit, 230
- zamkniętość, 24
- złożenie, 197
- znacznik czasu, 177
 - przepelnienie, 177

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Łańcuchy bloków: poznaj technologie kryptowalut od podszewki!

Kryptowaluty, bitcoin i łańcuch bloków kojarzą się z mrocznym półświatkiem, pełnym tajemnic środowiskiem przestępczym lub w ostateczności z buntem przeciw wszechmocnemu systemowi i politykom. Wiele publikacji, stwarzających pozory rzetelności, ukazuje te technologie jako źródło łatwych pieniędzy, rozbudzając ludzkie lęki i chciwość. Tymczasem łańcuch bloków jest wymagającą technologią o wyjątkowym potencjale. Nie ma w niej dróg na skróty ani gotowych rozwiązań. Aby zrozumieć łańcuch bloków, nie wystarczy lektura specyfikacji czy analitycznych opracowań. Trzeba samodzielnie zaprogramować podstawowe elementy aplikacji działającej na łańcuchu bloków.

Ta książka jest najskuteczniejszym sposobem na poznanie technologii bitcoina oraz łańcucha bloków przez programowanie. Dzięki niej zrozumiesz matematyczne podstawy protokołu bitcoin, zasady pracy z łańcuchem bloków i transakcjami, a także poznasz szczegóły najnowszych rozszerzeń tego protokołu. Nauczysz się zasad kryptografii klucza publicznego oraz sposobów przechowywania i przesyłania zdefiniowanych prymitywów kryptograficznych. Zapoznasz się z komunikacją sieciową w protokole bitcoin oraz z metodami pobierania i przesyłania danych do węzłów przechowujących łańcuch bloków. Zrozumienie prezentowanych treści okaże się łatwiejsze dzięki licznym ćwiczeniom praktycznym.

W tej książce:

- przetwarzanie transakcji bitcoinowych
- podstawy języka kontraktów inteligentnych Script
- programowanie rozliczeń z użyciem bitcoina
- zabezpieczanie łańcucha bloków
- techniki kryptograficzne, w tym prymitywy kryptograficzne

Jimmy Song jest doświadczonym programistą i współtwórcą wielu startupów. Od 2014 roku w pełni poświęca się bitcoinowi — bierze udział w wielu związanych z nim projektach open source, takich jak Armory, Bitcoin Core, btcd czy pycoin. Wykłada programowanie w protokole bitcoin na Uniwersytecie Teksańskim.

Helion 

 helion.pl

 **HELION SA**
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!

SZKOLENIA



AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-283-5923-9



INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł