فصل پنجم

تراکنش‌ها

تراکنش‌ها، قلب بیت‌کوین هستند. به بیان ساده‌تر، تراکنش‌ها، انتقال مقدار از یک موجودیت به موجودیت دیگر را سبب می‌شوند. در این فصل، به بررسی دقیق‌تراکنش و اجزا تشکیل‌دهنده‌ی آن می‌پردازیم.

اجزای تراکنش

در بالاترین سطح، تراکنش‌ها شامل ٤ جزء اساسی هستند:

۱. نسخه[1]: نشان می‌دهد که تراکنش از چه ویژگی‌های اضافه‌ی دیگری استفاده می‌کند.
۲. ورودی‌ها: نشان‌دهنده‌ی این است که چه بیت‌کوین‌هایی خرج می‌شوند.
۳. خروجی‌ها: نشان می‌دهد، بیت کوین‌ها کجا می‌روند.
٤. قفل زمانی[2]: مشخص می‌کند که تراکنش‌ها چه زمانی معتبر شناخته خواهند شد[3]. نمونه‌ای از تراکنش در شکل (۱–۵) نشان داده شده است، که بخش‌های مختلفِ تشکیل‌دهنده‌ی تراکنش به ترتیب نسخه، ورودی‌ها، خروجی و قفل زمانی است.

[1] Version
[2] Lock time
[3] ممکن است تراکنش های انجام شده در یک زمان آتی معین، اعمال شوند. این بدان معنی است که تا فرارسیدن آن زمان، تراکنش های انجام شده قفل می گردند.

شکل ۵-۱: نمایش هگزادسیمال تراکنش معمولی که هر بخش با رنگ متفاوتی نشان داده شده است

با در نظر داشتن این موضوع، کلاس تراکنش که Tx نامیده می‌شود به شکل زیر ایجاد می‌گردد:

```
class Tx:

    def __init__(self, version, tx_ins, tx_outs, locktime,
testnet=False):
        self.version = version
        self.tx_ins = tx_ins
        self.tx_outs = tx_outs
        self.locktime = locktime
        self.testnet = testnet

    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):
        '''Human-readable hexadecimal of the transaction
hash'''
        return self.hash().hex()

    def hash(self):
        '''Binary hash of the legacy serialization'''
        return hash256(self.serialize())[::-1]
```

1
2
3
4

۱. از آنجا که input و output دو اصطلاح رایج هستند، نوع و مقدار آن‌ها باید تعیین گردد.
۲. به منظور تعیین اعتبار تراکنش باید مشخص شود که تراکنش در کدام شبکه قرار دارد.
۳. Id همان شناسه‌ای است که می‌توان بر مبنای آن بلوک‌ها را جست‌وجو کرده و نمایش داد. این شناسه درواقع Hash256 تراکنش در قالب هگزادسیمال است.
٤. تابع هش، الگوریتم hash256 است که به روش Little-Endian پیاپی‌سازی می‌شود. توجه نمایید که متد `serialize` هنوز پیاده‌سازی نشده است.

در ادامه‌ی این فصل، تجزیه‌ی[۱] تراکنش بررسی می‌شود. در این راستا می‌توان کد تجزیه تراکنش را به‌صورت زیر نوشت:

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

1
2

۱. این متد[۲] باید یک *متد کلاس*[۳] باشد زیرا پیاپی سازی، نمونه جدیدی از یک شیء `Tx` را برمی‌گرداند.
۲. در اینجا فرض شده است که متغیر `serialization`، آرایه‌ای از بایت‌ها است.

روش فوق، ممکن است برای تعداد کمِ تراکنش‌ها مناسب باشد، اما اگر تعداد آن‌ها خیلی زیاد باشد، ایده‌آل آن است که بتوانیم جریان[٤] آن‌ها را تجزیه کنیم. روش اخیر این امکان را می‌دهد، که قبل از تجزیه‌ی تراکنش‌ها، نیازی به کل تراکنش‌ها به‌صورت یکجا نداشته

---

[۱] Parsing

[۲] به توابعی که درون یک کلاس پیاده سازی می شوند، متد گفته می شود.

[۳] Method class

[۴] Sream

باشیم. این روش در عمل کارآمدتر خواهد بود. کد اصلی تجزیه‌ی تراکنش می‌تواند به‌صورت زیر باشد:

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

1

۱. متد read، این اجازه را می‌دهد که تجزیه‌ی داده‌ها بدون نیاز به انتظار برای درگاه‌های I/O (ورودی/خروجی) انجام شود[۱].

این کار از منظر مهندسی سودمند است، زیرا جریان داده می‌تواند اتصال یک سوکت[۲] در شبکه یا دستگیره‌ی[۳] یک فایل باشد. می‌توان به‌جای منتظر ماندن برای انتقال یا خواندن همه چیز، بلافاصله شروع به تجزیه‌ی جریان داده نمود. این روش قادر خواهد بود هر نوع جریان داده‌ای را پذیرفته و شیء Tx موردنیاز را برگرداند.

نسخه

شماره‌ی نسخه‌ی تراکنش (شکل (۵-۱)) نشان می‌دهد که باید انتظار دریافت چه داده-هایی یا چه ویژگی‌هایی را داشته باشیم، به عنوان مثال، زمانی که کاربر از ویندوز ۳/۱ استفاده می‌کند، شماره‌ی نسخه (یعنی ۳/۱) خیلی متفاوت با نسخه‌ی ویندوز ۸ یا ۱۰ است. کاربر فقط می‌تواند مشخص کند که در حال استفاده از ویندوز است و نسخه‌ی ویندوز پس از اجرای سیستم‌عامل برای وی مشخص خواهد کرد که این نسخه چه ویژگی‌هایی دارد و در مقابل چه APIهایی را می‌تواند مورد استفاده قرار دهد.

---

[۱] به این روش، تجزیه روی هوا (on the fly) گفته می شود.

[۲] Socket

[۳] Handle

```
01000000
01813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

شکل ۵-۲: نسخه‌ی تراکنش

مشابه مثال فوق، تراکنش‌های بیت کوین نیز دارای شماره‌ی نسخه هستند. در بیت کوین عموماً از نسخه‌ی ۱ استفاده می‌شود، اما استفاده از نسخه‌ی ۲ هم در مواردی مرسوم است (تراکنش‌هایی که از opcode موسوم به OP_CHECKSEQUENCEVERIFY استفاده می‌کنند و در BIP0112 تعریف شده‌اند).

در سیستم هگزادسیمال نسخه را با 0100000 نشان می‌دهند که به نظر نمی‌رسد شبیه یک باشد. اگر به عنوان یک عدد صحیح Little-Endian تفسیر شود، درواقع همان عدد ۱ است.

ورودی‌ها

هر ورودی به خروجی تراکنش قبلی اشاره می‌کند (شکل ۵-۳). البته این موضوع نیاز به توضیح بیشتر دارد و به‌صورت واضح از ابتدا مشخص نیست.

```
01000000
01813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

شکل ۵-۳: ورودی تراکنش

درواقع ورودی‌های بیت‌کوین، خروجی‌های خرج نشده‌ی تراکنش‌های قبلی را خرج می‌کنند. به این معنی که ابتدا نیاز است بیت کوین‌ها دریافت شده و سپس برای آنچه که

مدنظر است هزینه گردند. درواقع نمی‌توان بدون دریافت اولیه بیت‌کوین، آن‌ها را خرج نمود. هر ورودی دو ویژگی خواهد داشت:

۱. یک مرجع به بیت کوین‌هایی که قبلاً دریافت شده است.
۲. اثبات اینکه این بیت‌کوین‌ها متعلق به کاربری هستند که می‌خواهد آن‌ها را خرج کند.

در مرحله‌ی دوم از الگوریتم ECDSA استفاده می‌شود. برای جلوگیری از جعل بیت کوین توسط کاربران، اغلب ورودی‌ها شامل امضاهایی هستند که تنها دارنده یا دارندگان کلید خصوصی می‌توانند آن‌ها را خرج کنند.

بخش ورودی‌ها می‌تواند شامل بیش از یک ورودی باشد. این دقیقاً مشابه هزینه کردن یک اسکناس ۱۰۰ دلاری یا استفاده از دو اسکناس ۵۰ و ۲۰ دلاری برای یک وعده غذای ۷۰ دلاری است. که در حالت اول نیازمند یک ورودی و در حالت دوم نیازمند دو ورودی است. شرایطی وجود دارد که تعداد ورودی‌ها می‌تواند بیشتر نیز باشد. در مثال ذکر شده، می‌توان ۷۰ دلار را با ١٤ اسکناس ۵ دلاری یا حتی ۷۰۰۰ پنی نیز پرداخت کرد. که در این شرایط تعداد ورودی‌ها ١٤ یا ۷۰۰۰ ورودی مشابه است. همان‌طور که در شکل ۵-٤ مشخص شده، تعداد ورودی‌ها، بخش بعدی تراکنش است.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

شکل ۵-٤: تعداد ورودی‌های یک تراکنش

در رشته‌ی فوق، بایت 01، به این معنی است که این تراکنش یک ورودی دارد. توجه به این نکته ضروری است، که تنها یک بایت معرف تعداد ورودی‌ها نیست، چون یک بایت ۸ بیت است که ورودی‌های بیشتر از ۲۵۵ را نمی‌توان با یک بایت توصیف کرد.

در این حالت است که *varints* مورد استفاده قرار می‌گیرد که کوتاه شده‌ی variable integer است. در این روش اعداد صحیح در محدوده‌ی ۰ تا ۱ – $۲^{۶۴}$ با یک یا چند بایت بیان (کد) می‌شوند. البته برای تعداد ورودی‌ها می‌توان ۸ بایت را ذخیره کرد، اما باعث عدم استفاده‌ی مفید از حافظه می‌شود؛ بخصوص اگر انتظار داشته باشیم که تعداد ورودی‌ها نسبتاً کم، مثلاً در حدود ۲۰۰ یا کمتر باشد. در مورد تراکنش‌های عادی این شرایط وجود دارد. بنابراین، استفاده از *varints* به صرفه‌جویی در فضا کمک می‌کند. روش کار *varints* به‌صورت زیر است:

۱. اگر عدد کمتر از ۲۵۳ باشد، با یک بایت کد می‌شود (مثلاً 0x64→100).
۲. اگر عدد بین ۲۵۳ و ۱ – $۲^{۱۶}$ باشد، با بایت ۲۵۳ یا (fd) آغاز شده و سپس عدد در دو بایت به‌صورت Little-Endian کد می‌شود (مثلاً 0xfdff00 → 255 و 0xfd2b02→ 255).
۳. اگر عدد بین $۲^{۱۶}$ و ۱ – $۲^{۳۲}$ باشد، با بایت ۲۵٤ یا (fe) آغاز شده و سپس عدد در ٤ بایت به‌صورت Little-Endian کد می‌شود (مثلاً 0xfe7f110100 ⟶ 70015).
٤. اگر عدد بین $۲^{۳۲}$ و ۱ – $۲^{۶۴}$ باشد، با بایت ۲۵۵ یا (ff) آغاز شده و سپس عدد در ۸ بایت به‌صورت little-Endian کد می‌شود (مثلاً 0xffd6dc7ed3e60100000 ⟶ 18005558675309).

کد پایتون روش *varints* در ادامه آورده شده است:

```
def read_varint(s):
    '''read_varint reads a variable integer from a stream'''
    i = s.read(1)[0]
    if i == 0xfd:
        # 0xfd means the next two bytes are the number
        return little_endian_to_int(s.read(2))
    elif i == 0xfe:
        # 0xfe means the next four bytes are the number
        return little_endian_to_int(s.read(4))
    elif i == 0xff:
        # 0xff means the next eight bytes are the number
        return little_endian_to_int(s.read(8))
    else:
        # anything else is just the integer
        return i
```

```
def encode_varint(i):
    '''encodes an integer as a 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('integer too large: {}'.format(i))
```

`read_varint` یک متغیر صحیح را از یک جریان داده خوانده و عدد صحیحِ کد شده را برمی‌گرداند. `encode_varint` برعکس عمل خواهد کرد، یعنی یک عدد صحیح را دریافت کرده و بایت نتیجه را مبتنی بر روش *varint* برمی‌گرداند. هر ورودی، از ٤ بخش تشکیل شده است:

١. شناسه‌ی[١] تراکنش قبلی
٢. نمایه‌ی[٢] تراکنش قبلی
٣. اسکریپت امضا (ScriptSig)
٤. توالی[٣]

همان‌طور که توضیح داده شد، هر ورودی به خروجیِ تراکنش قبلی اشاره دارد. شناسه‌ی تراکنش قبلی، درواقع همان محتوای تراکنش قبلی است که به‌صورت hash256 درآمده است. این تابع هش، تراکنش قبلی را به‌طور منحصربه‌فرد تعریف می‌کند، زیرا احتمالِ تصادمِ[٤] هش غیرممکن است. همان‌طور که مشاهده خواهد شد، حداقل تعداد خروجی هر تراکنش یک است اما این تعداد بیشتر نیز می‌تواند باشد.

بنابراین، باید دقیقاً مشخص شود که طی یک تراکنش چه خروجی‌ای خرج می‌شود (این خروجی در نمایه‌ی تراکنش قبلی ثبت شده است). توجه به این نکته ضروری است

---

١ ID
2 Index
3 Sequence
٤ collision

که شناسه‌ی تراکنش قبلی ۳۲ بایت و نمایه‌ی تراکنش قبلی، ٤ بایت و هر دو مبتنی بر استاندارد little-endian هستند.

ScriptSig می‌بایست با زبانِ قرارداد هوشمند[۱] بیت کوین یعنی اسکریپت[۲]، تعامل داشته باشد. این موضوع به‌طور دقیق‌تری در فصل ٦ مورد بررسی قرار گرفته است، اما در حال حاضر تنها به ذکر این نکته بسنده می‌کنیم که عملکرد ScriptSig مانند باز کردن یک صندوقچه است. این عمل تنها توسط مالک خروجیِ تراکنش قابل انجام است. فیلد Scriptsig فیلدی با طول متغیر است. یک فیلد با طول متغیر، این الزام را به وجود می‌آورد که طول آن دقیقاً مشخص گردد (متغیر بودنِ فیلد اشاره به طول متغیر آن دارد).

بخش *توالی*[۳] همان مفهومی است که ساتوشی از آن به عنوان، «معاملات فرکانس بالا» به همراه بخش *قفل زمانی* یاد کرده است. اما امروزه به همراه Replace-By-Fee (RBF) و OP_CHECKSEQUENCEVERIFY استفاده می‌شود. *توالی* ٤ بایت است که به روش little- Endian کد می‌شود. بخش‌های مختلفِ یک ورودی به‌صورت رنگی و با رنگ‌های مختلف در شکل (۵-۵) نشان داده شده‌اند که به ترتیب: شناسه‌ی تراکنش قبلی، نمایه‌ی تراکنش قبلی، ScriptSig و Sequence هستند.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

شکل ۵-۵: بخش‌های مختلف یک ورودی تراکنش

توالی و قفل زمانی

در ابتدا، ساتوشی می‌خواست از فیلدهای *توالی* و *قفل زمانی* برای مفهومی به نام «معاملات فرکانس بالا» استفاده کند. آنچه ساتوشی تصور می‌کرد روشی برای انجام بازپرداخت به

[۱] Smart contract

[۲] Script

[۳] Sequence

شخص دیگری بدون نیاز به انجام تعداد بالایی تراکنش‌های زنجیره‌ای بود. به عنوان مثال، اگر آلیس بخواهد به دلیلی، تعداد x بیت کوین را به باب پرداخت کند و سپس باب، تعداد y بیت کوین را به دلیل دیگری به آلیس بپردازد (مثلاً اگر x بزرگ‌تر ازy باشد)، پس آلیس می‌تواند به‌جای اینکه دو تراکنش جداگانه در زنجیره انجام دهد، فقط y - x را به باب بپردازد.
ایده‌ای که ساتوشی در این زمینه داشت شامل یک دفترکل فرعی کوچک بود که در آن، به‌صورت مدام تراکنش‌های بین دو طرف درگیرِ معامله، به روز رسانی می‌شوند و این روش جایگزین ثبت معاملات در دفترکل اصلی بود. هدف ساتوشی این بود که از فیلدهای *توالی* و *قفل زمانی* برای به روز رسانیِ معاملات فرکانس بالا در زمان پرداخت جدید بین دو طرف استفاده کند. تراکنش‌های مربوط به *معاملات*، دارای دو ورودی (یکی از آلیس و دیگری از باب) و دو خروجی (یکی به آلیس و دیگری به باب) هستند. تراکنش‌های *معاملاتی* با *توالی* 0 و با *قفل زمانی* درازمدت[1] شروع می‌شوند (مثلاً ۵۰۰ بلوک از الآن، که تا ۵۰۰ بلوک آتی معتبر است). این پایه‌ای برای تراکنشی است که در آن آلیس و باب همان مبلغی را که در این تراکنش منظور کرده‌اند، دریافت می‌کنند.
پس از تراکنش اول، که آلیس تعداد x بیت کوین را به باب می‌پردازد، *توالی* هر ورودی ۱ خواهد بود. پس از تراکنش دوم، که باب تعداد y بیت کوین را به آلیس پرداخت می‌کند، *توالی* هر ورودی ۲ خواهد بود. با استفاده از این روش، می‌توان پرداخت‌های زیادی را مادامی که قبل از معتبر بودن *قفل زمانی* اتفاق افتاده باشد، در یک تراکنشِ تک زنجیره‌ای فشرده کرد. متأسفانه، تقلب برای یک استخراج‌کننده بسیار آسان است. در مثال ذکر شده، باب می‌تواند یک استخراج‌کننده باشد. او می‌تواند تراکنش *معاملاتی* به روز شده با *توالی* شماره‌ی ۲ را نادیده بگیرد و تراکنش *معاملاتی* را با *توالی* شماره‌ی ۱ ۱ استخراج کند، و آلیس را با y بیت کوین فریب دهد.
بعدها طراحی بسیار بهتری مبتنی بر «کانال‌های پرداخت[2]» ارائه شد، که پایه و اساس شبکه‌های سبک‌تر قرار گرفت.

[1] Far-away

[2] Payment channels

اکنون با معرفی بخش‌های مختلف یک تراکنش می‌توان یک کلاس TxIn را به‌صورت زیر نوشت:

```
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:
            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,
        )
```

۱. به‌صورت پیش‌فرض مقدار اولیه ScriptSig بدون مقدار در نظر گرفته می‌شود.

یادآوری چند نکته ضروری به نظر می‌رسد. اول اینکه مقدار هر یک از ورودی‌ها مشخص نیست. اطلاعی از مقدار خرج شده در دسترس نیست مگر این‌که آن را در زنجیره بلوکی تراکنش(های) انجام شده جستجو کنیم. علاوه بر این، حتی نمی‌دانیم که تراکنش جعبه‌ی درستی را باز می‌کند یا خیر. بنابراین، می‌توان گفت، بدون اطلاع در مورد تراکنش‌های قبلی، نمی‌توان در مورد ورودی‌ها صحبت کرد. هر گره باید تائید کند که این تراکنش جعبه‌ی درست را باز می‌کند و بیت‌کوین‌های ناموجود را خرج نمی‌کند. نحوه انجام این کار در فصل ۷ بیشتر مورد بحث قرار گرفته است.

تجزیه‌ی اسکریپت

در فصل ٦، به‌طور کامل در مورد اسکریپت صحبت خواهد شد، اما اکنون، نشان می‌دهیم که اسکریپت به عنوان یک شیء Script در پایتون، چگونه از یک سیستم عددی هگزادسیمال دریافت می‌شود:

```
>>> from io import BytesIO
>>> from script import Script
>>> script_hex =
('6b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031cc
f\
```

1

```
cf21320b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f
67801c31967743a9c8\
e10615bed01210349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd
531d2e213bf016b278\
a')
>>> stream = BytesIO(bytes.fromhex(script_hex))
>>> script_sig = Script.parse(stream)
>>> print(script_sig)
3045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf213
20b0277457c98f0220\
7a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e1061
5bed01 0349fc4e631\
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a
```

۱. کلاس `Script` در فصل ٦ با جزئیات بیشتری بررسی خواهد شد، اما به‌طور کلی، متد `Script.parse` شیء موردنیاز کاربر را ایجاد می‌کند.

خروجی‌ها

همان‌طور که در فصل قبل نیز گفته شد، خروجی‌ها مشخص می‌کنند که بیت‌کوین‌ها به کجا ارسال شوند. هر تراکنش یک یا چند خروجی خواهد داشت. اما چرا یک تراکنش ممکن است چند خروجی داشته باشد؟ چون ممکن است یک فرد، هم‌زمان برای چند نفر که درخواست بیت‌کوین کرده‌اند پرداخت داشته باشد. شکل (٥-٦) تعداد خروجی-های یک تراکنش را نشان می‌دهد. در این رشته دو خروجی داریم.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

شکل ٥-٦: تعداد خروجی‌های یک تراکنش معمولی

هر خروجی ۲ بخش دارد، ۱- *مقدار* و ۲- ScriptPubKey

۱. *مقدار*: *مقدار* بیت‌کوینی که جابجا می‌شود می‌تواند برحسب ساتوشی یا یک صد میلیونیمِ بیت‌کوین باشد که همین باعث می‌شود، بتوان بیت کوین را تا یک سیصدم پنی در USD (دلار آمریکا) خرد کرد. با توجه به حداکثر تعداد بیت

کوین که ۲۱ میلیون در نظر گرفته شده است، معادل ساتوشی آن ۲،۱۰۰،۰۰۰،۰۰۰،۰۰۰،۰۰۰ (۲،۱۰۰ تریلیون) خواهد بود. این عدد بزرگ‌تر از 2^{32} است، درنتیجه باید در ٦٤ بیت یا ۸ بایت ذخیره شود که روشِ ذخیره‌سازی آن little-Endian است.

۲. ScriptPubKey: همانند ScriptSig مبتنی بر زبان قراردادهای هوشمند بیت کوین (اسکریپت) نوشته می‌شود. ScriptPubKey همانند یک جعبه‌ی قفل‌شده است که تنها دارنده‌ی کلید خصوصی می‌تواند آن را باز کند و یک راه مطمئن و امن برای دریافت سهام و اندوخته از دیگران است. مشابه ScriptSig و آنچه پیش‌تر توضیح داده شد، ScriptPubKey نیز از فیلد *طول متغیر* و تکنیک *varints* استفاده می‌کند که در فصل ٦ در مورد آن توضیحات مبسوط ارائه خواهد شد.

در شکل (۵-۷) خروجی یک تراکنش به‌طور کامل نشان داده شده است.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

شکل ۵-۷: خروجی یک تراکنش در سیستم عددی هگزادسیمال

مجموعه‌ی UTXO

UTXO[1] مخفف تراکنش خروجیِ خرج نشده است. به مجموعه خروجی‌های خرج نشده‌ی یک تراکنش در هر لحظه، UTXO گفته می‌شود. از آنجایی که این خروجی‌ها در هر لحظه میزان بیت‌کوین آماده برای خرج شدن را نشان می‌دهد، بسیار مهم هستند. درواقع UTXOها بیت‌کوین‌هایی هستند که در چرخه قرار دارند. گره‌های کاملِ شبکه، باید مجموعه‌ی UTXO را ردیابی کرده با نگهداری نمایه‌های[2] آن‌ها، سرعت تائید تراکنش‌های جدید را افزایش دهند.

[1] Unspent Transaction Outputs: UTXO

[2] Indexed

به عنوان مثال، با جستجوی خروجی تراکنش قبلی در مجموعه‌ی UTXO به‌راحتی می‌توان از دوباره خرج کردن[1] جلوگیری کرد. اگر ورودی یک تراکنش جدید، از خروجی تراکنشی استفاده کند که در مجموعه‌ی UTXO قرار ندارد، درواقع به عنوان تلاشی برای دوباره خرج کردن و یا یک خروجی ناموجود تلقی شده و در هر دو حالت، غیر معتبر است. بنابراین داشتن مجموعه‌ی UTXO در تائید صحت تراکنش‌ها بسیار ضروری است. همان‌طور که در فصل ٦ مشاهده خواهد شد، برای اعتبارسنجی تراکنش‌ها، باید مقدار و ScriptPubKey از خروجی تراکنشِ قبلی جستجو شود، بنابراین داشتن این UTXO ها می‌تواند فرایند تائید اعتبار تراکنش‌ها را تسریع کند.

در ادامه کد پایتون کلاس `TxOut` مشاهده خواهد شد:

```
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)
```

قفل زمانی

قفل زمانی راهی برای ایجاد تأخیر در تراکنش است. یک تراکنش با قفل زمانی ٦٠٠٠٠٠ نمی‌تواند، تا بلوک ٦٠٠٠٠١ وارد زنجیره بلوکی شود. در ابتدا از این قفل زمانی برای انجام معاملات فرکانس بالا استفاده می‌شد که البته از امنیت بالایی برخوردار نبود. اگر قفل زمانی بزرگ‌تر یا برابر با ٥٠٠/٠٠٠/٠٠٠ باشد، به آن unix timestamp می‌گویند و اگر کمتر از این مقدار باشد یک شماره بلوک است. به این ترتیب، تراکنش‌ها می‌توانند امضا شوند اما تا زمانی که به unix time خاصی یا شماره بلوک مشخصی نرسیده باشند، قابل خرج شدن نیستند.

چه زمانی قفل زمانی نادیده گرفته می‌شود؟

[1] Double spend

قفل زمانی برای *توالی* با مقدار ffffffff برای هر ورودی نادیده گرفته می‌شود.

در شکل (۵-۸) قفل زمانیِ یک تراکنش در سیستم عددی هگزادسیمال نشان داده شده است.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

شکل ۵-۸: قفل زمانی

مشکل اصلی استفاده از قفل زمانی این است که گیرنده‌ی تراکنش مطمئن نیست که با فرارسیدن قفل زمانی انجام تراکنش خوب است یا نه. دقیقاً همانند چکی که تاریخ آن فرا رسیده است اما ممکن است برگشت بخورد. فرستنده می‌تواند ورودی‌ها را قبل از فرارسیدن *قفل زمانی* خرج کرده باشد، بنابراین تراکنش هنگام فرارسیدن *قفل زمانی*، دیگر معتبر نیست. موارد استفاده، قبل از BIP0065 محدود بود اما این پروتکل با معرفی OP_CHECKLOCKTIMEVER توانست مانع خرج شدن خروجی، قبل از رسیدن *قفل زمانی* شده و مشکلات آن را رفع کرد.

کد کردن تراکنش‌ها

در بخش‌های قبلی، تجزیه‌ی تراکنش‌ها بررسی شد، در ادامه به پیاپی‌سازی تراکنش پرداخته می‌شود، که از کلاس TxOut شروع می‌کنیم:

```
class TxOut:
...
    def serialize(self):                                    (1)
        '''Returns the byte serialization of the transaction
output'''
        result = int_to_little_endian(self.amount, 8)
        result += self.script_pubkey.serialize()
        return result
```

۱. در اینجا هدف این است که شیء کلاسِ TxOut پیاپی‌سازی شود.

به همین روش می‌توان کلاس TxIn را پیاپی‌سازی نمود:

```
class TxIn:
...
    def serialize(self):
        '''Returns the byte serialization of the transaction
input'''
        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
```

و در آخر کلاس Tx:

```
class Tx:
...
    def serialize(self):
        '''Returns the byte serialization of the
transaction'''
        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
```

برای پیاپی‌سازی Tx از متدهای پیاپی‌ساز TxIn و TxOut استفاده شده است. توجه به این نکته ضروری است که کارمزد تراکنش در جایی مشخص نشده است! به این دلیل که کارمزد، همان‌طور که در بخش بعدی شرح داده شده است، مقداری ضمنی است.

کارمزد تراکنش

یکی از قوانین *اجماع*[1] در بیت‌کوین برای هر تراکنش *غیر سکه پایه*[2] این است که باید مجموع ورودی‌ها بیشتر یا برابر خروجی‌ها باشند که در فصل ۹ علت آن بررسی می‌شود. ممکن است تعجب کنید که چرا ورودی‌ها و خروجی‌ها نمی‌توانند کاملاً برابر باشند؟ علت آن است که اگر هزینه‌ی هر تراکنش صفر باشد، انگیزه‌ای برای استخراج‌کنندگان

[1] Consensus

[2] Non-Coinbase

وجود نخواهد داشت که آن تراکنش را در بلوک ثبت کنند. درواقع پرداخت کارمزد، تشویقی برای استخراج‌کننده‌هاست تا با دریافت آن، تراکنش را در زنجیره‌ی بلوکی ثبت کنند، زیرا تراکنش‌هایی که هنوز وارد زنجیره‌ی بلوکی نشده‌اند و به آن‌ها تراکنش‌های mempool گفته می‌شود، بخشی از زنجیره بلوکی نبوده و نهایی نشده‌اند.

کارمزد از تفریق مجموع ورودی‌ها از مجموع خروجی‌ها به دست می‌آید. کارمزد توسط استخراج‌کننده‌ها، دریافت می‌گردد. این تفاوت همان چیزی است که استخراج‌کننده می‌تواند نگه دارد. از آنجا که در ورودی، فیلد *مقدار* وجود ندارد، باید مقدار را جستجو کنیم. این جست‌وجو نیاز دسترسی به زنجیره بلوکی و UTXO را نشان می‌دهد. دسترسی به کل زنجیره‌ی بلوکی توسط گره‌ی کامل امکان‌پذیر خواهد بود، چرا که، دفتر کل را به‌صورت محلی در اختیار دارد. اما اگر یک گره‌ی کامل، اجرا نشود، باید از یک موجودیت دیگر درخواست کنیم که این اطلاعات را در اختیارمان قرار دهد. در ادامه کلاس TXfetcher برای محاسبه‌ی کارمزد مشاهده خواهد شد:

```
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('unexpected response:
{}'.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:
                raise ValueError('not the same id: {} vs
{}'.format(tx.id(), tx_id))
            cls.cache[tx_id] = tx
        cls.cache[tx_id].testnet = testnet
        return cls.cache[tx_id]
```

۱. شناسه‌ی تراکنش به منظور کسب اطمینان از اینکه این شناسه همان شناسه‌ی مورد انتظار کاربر است، چک می‌شود.

سؤالی که مطرح می‌شود این است، که چرا فقط بخشی از تراکنش را مد نظر قرار نداده و کل یک تراکنش در نظر گرفته می‌شود؟ دلیل این کار این است که نمی‌خواهیم شخص ثالثی وارد تراکنش‌ها شود. با دریافت کل یک تراکنش، به‌راحتی می‌توان شناسه‌ی تراکنش (hash256 محتوای آن) را تائید کرد و اطمینان حاصل کرد که در حال دریافت تراکنشی هستیم که آن را درخواست کرده‌ایم و این فرآیند تا دریافت کل یک تراکنش، ممکن نخواهد شد.

چرا نیاز است که اعتماد به شخص ثالث به حداقل برسد؟

همان‌طور که نیک سابو[۱] در مقاله‌ی اصلی خود با عنوان «اشخاص ثالثِ مورد اعتماد، حفره‌های امنیتی هستند» نوشت، اعتماد به اشخاصِ ثالث برای ارائه‌ی داده‌های صحیح، روش امنیتی مناسبی نیست. شخص ثالث ممکن است در زمان کنونی رفتار خوبی داشته باشد، اما هرگز نمی‌توان اطمینان داشت که چه زمانی ممکن است هک شود یا به یک کارمند سرکش تبدیل شود یا سیاست‌های مغایر با منافع کاربر را در پیش گیرد. چیزی که بیت‌کوین را ایمن می‌کند اعتماد نیست، بلکه تأیید داده‌هایی است دریافت می‌کنیم.

اکنون می‌توان روش مناسبی را در TxIn برای واکشی تراکنش قبلی و روش‌های به دست آوردن مقدار خروجیِ تراکنش قبلی و ScriptPubKey ایجاد نمود (مورد دوم در فصل ٦ استفاده می‌شود).

```
class TxIn:
...
```

[۱] Nick Szabo

```
    def fetch_tx(self, testnet=False):
        return TxFetcher.fetch(self.prev_tx.hex(),
testnet=testnet)

    def value(self, testnet=False):
        '''Get the output value by looking up the tx hash.
        Returns the amount in satoshi.
        '''
        tx = self.fetch_tx(testnet=testnet)
        return tx.tx_outs[self.prev_index].amount
    def script_pubkey(self, testnet=False):
        '''Get the ScriptPubKey by looking up the tx hash.
        Returns a Script object.
        '''
        tx = self.fetch_tx(testnet=testnet)
        return tx.tx_outs[self.prev_index].script_pubkey
```

محاسبه‌ی کارمزد

حال که متد value را در کلاس TxIn داریم، این امکان وجود دارد که تعداد بیت کوین‌ها در هر ورودیِ تراکنش، مشخص باشد و از این رو بتوان کارمزد تراکنش را محاسبه نمود.