





Introduction to Bitcoin Scripting



Konstantinos Karasavvas
R&D Blockchain Initiative, University of Nicosia



Agenda

Transactions

Signatures

Scripting

Creating Transactions

Segregated Witness (Segwit)

Data Storage

Timelocks



Transactions

Transactions and Scripting

- Transaction
 - 1+ inputs
 - 1+ outputs
 - UTXOs -- always fully spend
- UTXOs have conditions on how to spend them
 - to unlock one needs to satisfy them
 - conditions are defined with a scripting language
- When we sent funds
 - we specify the locking script (scriptPubKey)
- When we spend funds
 - we specify the unlocking script (scriptSig)

Transaction Types

- Scripting language
 - several operators
 - stack-based language (reverse polish notation)
 - no programming loops
- P2PKH
 - pay to a Bitcoin address
- P2PK
 - obsolete
- P2SH
 - arbitrarily complex scripts

P2PKH (1)

tx (ffdfa1de5...)

...

Value: 2
scriptPubKey: opcodes
(include pubkey
Hash160 of 1Alice
bitcoin address)

...

Alice "owns" bitcoin address 1Alice and wants to sent 1BTC to Bob.

This is the standard transaction type to transfer bitcoins.

transaction

Input(s)

Previous tx: fdfa1de5...
Index: 0
scriptSig: (Alice's sig) +
(Alice's public key)

....

Output(s)

Value: 1
scriptPubKey: opcodes
(include pubkey Hash160 of
1Bob bitcoin address)

....

P2PKH (2)

- scriptPubKey (locking script):

```
OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG
```

- scriptSig (unlocking script):

```
<Signature> <PublicKey>
```

- PKHash (Base58Check decoding, removing version and checksum bytes)
- signature uses ECDSA algorithm with secp256k1 parameters
- Validation -> **scriptSig** + **scriptPubKey** = true

P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

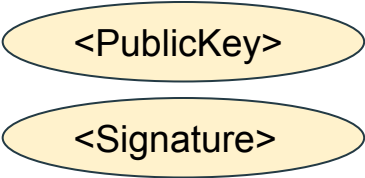
STACK:

`<Signature>`

P2PKH (3)

SCRIPT: <Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
 OP_EQUALVERIFY OP_CHECKSIG

STACK:



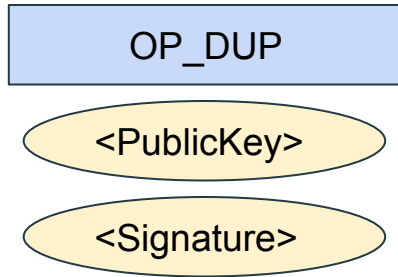
<PublicKey>

<Signature>

P2PKH (3)

SCRIPT: <Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
 OP_EQUALVERIFY OP_CHECKSIG

STACK:



P2PKH (3)

SCRIPT: <Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
 OP_EQUALVERIFY OP_CHECKSIG

RIPEND(SHA256(top item))

OP_HASH160

<PublicKey>

<PublicKey>

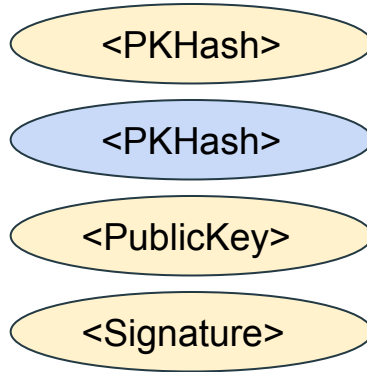
STACK:

<Signature>

P2PKH (3)

SCRIPT: <Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
 OP_EQUALVERIFY OP_CHECKSIG

STACK:



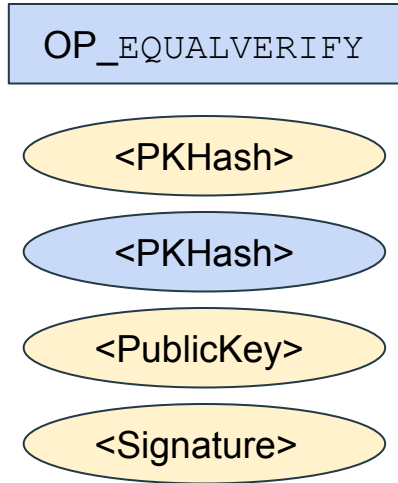
P2PKH (3)

SCRIPT: <Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
 OP_EQUALVERIFY OP_CHECKSIG

OP_EQUAL checks 2 top items
and replaces them with true or
false

OP_VERIFY checks top item
and if true removes it and if
false it terminates the script

STACK:



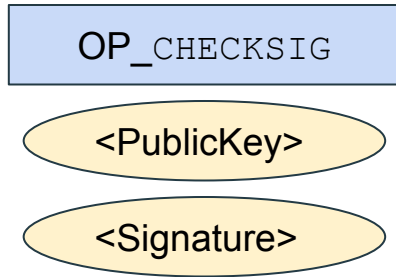
If the two <PKHash>es are
equal the <PublicKey>
provided was correct!

P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

OP_CHECKSIG consumes a public key and a signature. The Tx that we wish to spend is hashed and then the system validates the signature using the verified public key.

STACK:



Signatures Reminder:

The sender hashes parts of the Tx (the message), signs it and includes it in the new Tx.

The system then gets the Tx as well (the message) and hashes it. It then verifies that the signature and public key produce the same hash !

P2PKH (3)

SCRIPT: `<Signature> <PublicKey> OP_DUP OP_HASH160 <PKHash>
OP_EQUALVERIFY OP_CHECKSIG`

Success !

The system validated the ownership of the UTXO which is about to be spent.

STACK:



true

Example Transaction (1)

Let us examine the following mainnet transaction:

6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b

Transaction View information about a bitcoin transaction

[6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b](#)

[1Dj8woELooyfmsYH2ifPhP6vsqVTTpcaxJ](#)



[1524uZcTYKiECrKkYZwvzDoxRtc1mU23sM](#)

1.2946 BTC

6 Confirmations

1.2946 BTC

Summary

Size	191 (bytes)
Received Time	2017-07-06 08:50:58
Included In Blocks	474484 (2017-07-06 08:53:29 + 3 minutes)
Confirmations	6 Confirmations
Relayed by IP	176.58.96.62 (whois)

Inputs and Outputs

Total Input	1.2957 BTC
Total Output	1.2946 BTC
Fees	0.0011 BTC
Fee per byte	575.916 sat/B
Estimated BTC Transacted	1.2946 BTC

Example Transaction (2)

Transactions are transferred as a sequence of bytes.

```
$ ./bitcoin-cli getrawtransaction
```

```
6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b
```

```
01000000014655ecb69a2660ee381b5e2d8e616c97bde4144d9bf3aeb2b51442841  
47e1ba9000000006a47304402200b113ac8ff3699aa213055e3dcacea8509b7ffa3  
6d2cdc6a278bd16b371dcb9802206d3dcc6f0e9d99fe14e10f7f7fa806a88cfe7bb  
a20360deef9e74229a1d562f50121027f922a3403503d143404d2cf18df94899070  
673b4cdee3e08be3c8db7e6467aaffffffff012067b707000000001976a9142c142  
e0bc01f9cc4623f6b4613696d5c98b1141e88ac00000000
```

Example Transaction (3)

```
./bitcoin-cli decoderawtransaction 01000000014655ecb ... 41e88ac00000000
```

```
{  
  "txid": "6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b",  
  "hash": "6575caba0d157f150f6cd7d1c7577c6d03640f0a2a0f759ff1011fc1d18a4f8b",  
  "size": 191,  
  "vsize": 191,  
  "version": 1,  
  "locktime": 0,
```

Example Transaction (3)

```
"vin": [  
  {  
    "txid": "a91b7e14844214b5b2aef39b4d14e4bd976c618e2d5e1b38ee60269ab6ec5546",  
    "vout": 0,  
    "scriptSig": {  
      "asm":  
"304402200b113ac8ff3699aa213055e3dcacea8509b7ffa36d2cdc6a278bd16b371dcb9802206d3dcc6f0e9d99fe14e10f  
7f7fa806a88cfe7bba20360deef9e74229a1d562f5[ALL]  
027f922a3403503d143404d2cf18df94899070673b4cdee3e08be3c8db7e6467aa",  
      "hex":  
"47304402200b113ac8ff3699aa213055e3dcacea8509b7ffa36d2cdc6a278bd16b371dcb9802206d3dcc6f0e9d99fe14e1  
0f7f7fa806a88cfe7bba20360deef9e74229a1d562f50121027f922a3403503d143404d2cf18df94899070673b4cdee3e08  
be3c8db7e6467aa"  
    },  
    "sequence": 4294967295  
  }  
],
```

Example Transaction (3)

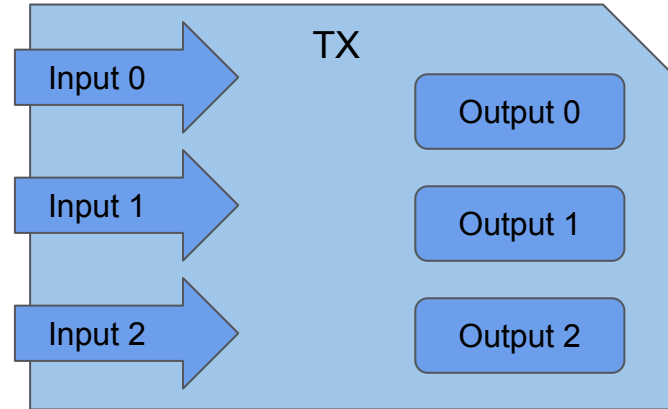
```
"vout": [  
  {  
    "value": 1.29460000,  
    "n": 0,  
    "scriptPubKey": {  
      "asm": "OP_DUP OP_HASH160 2c142e0bc01f9cc4623f6b4613696d5c98b1141e OP_EQUALVERIFY OP_CHECKSIG",  
      "hex": "76a9142c142e0bc01f9cc4623f6b4613696d5c98b1141e88ac",  
      "reqSigs": 1,  
      "type": "pubkeyhash",  
      "addresses": [  
        "1524uZcTYKiECrKkYZwvzDoxRtc1mU23sM"  
      ]  
    }  
  }  
]
```



Signatures

Signatures

- Spending UTXOs requires a signature
- The signature proves:
 - that the signer is the owner of the private key
 - the proof of authorization is undeniable
 - the parts of the tx that were signed cannot be modified after it has been signed
- Digital Signature Algorithm
 - ECDSA using DER serialization
- Hash type (SIGHASH)
 - 1 byte suffix
 - determines what is signed



SIGHASH flags

Flag	Value	Description
ALL	0x01	Signs all the inputs and outputs, protecting everything except the signature scripts against modification.
NONE	0x02	Signs all of the inputs but none of the outputs, allowing anyone to change where the satoshis are going unless other signatures using other signature hash flags protect the outputs.
SINGLE	0x03	The only output signed is the one corresponding to this input (the output with the same output index number as this input), ensuring nobody can change your part of the transaction but allowing other signers to change their part of the transaction. The corresponding output must exist or the value "1" will be signed, breaking the security scheme. This input, as well as other inputs, are included in the signature. The sequence numbers of other inputs are not included in the signature, and can be updated.

SIGHASH flags modifier: ANYONECANPAY

Flag	Value	Description
ALL ANYONECANPAY	0x81	Signs all of the outputs but only this one input, and it also allows anyone to add or remove other inputs, so anyone can contribute additional satoshis but they cannot change how many satoshis are sent nor where they go.
NONE ANYONECANPAY	0x82	Signs only this one input and allows anyone to add or remove other inputs or outputs, so anyone who gets a copy of this input can spend it however they'd like.
SINGLE ANYONECANPAY	0x83	Signs this one input and its corresponding output. Allows anyone to add or remove other inputs.



Scripting

Script

- Script is typically denoted with opcodes
 - compiled to a byte sequence
- Many opcodes
 - most unused
- Constants (OP_3, OP_8)
- Arithmetic operations (OP_ADD, OP_SUB)
- Boolean operators
- Cryptographic operators
- ... more later

Script: Example 1

Script: OP_3 OP_4 OP_ADD OP_5 OP_SUB

Hex: 53 54 93 55 94

Stack: []

[3]

[3,4]

[7]

[7,5]

[2]

Result is: 2

Script: Example 2

Script: OP_3 OP_4 OP_EQUAL OP_IF OP_5 OP_ELSE OP_10 OP_ENDIF

Hex: 53 54 87 63 55 67 60 68

Stack: []

[3]

[3,4]

[0]

[10]

Result is: 10

Script: Example 3

Just for completeness the hex sequence for a scriptPubKey of a P2PKH is:

```
scriptSig:      <Signature>          <PublicKey>
Hex:           49 11..ff 01          41 11..ff

scriptPubKey:  OP_DUP OP_HASH160 <PKHash> OP_EQUALVERIFY OP_CHECKSIG
Hex:           76          a9          14 11..ff          88          ac
```

Notice that values are *pushed* into the stack with OPs that are implied (not visible in the script).

Script: More examples

OP_TRUE

OP_FALSE

OP_NOT

OP_NOP

OP_RETURN <20 bytes in hex>

OP_ADD OP_SUB OP_2 OP_EQUAL

OP_HASH256 <32 bytes in hex> OP_EQUAL

2 <pubkey A> <pubkey B> <pubkey C> 3 OP_CHECKMULTISIG

What would unlock the
scriptPubKey's ?

Debugging Script

[Bitcoin Script IDE](#) or a command-line tool called, [btcdeb](#).

```
$ ./btcc OP_1 OP_2 OP_ADD  
515293
```

```
$ ./btcdeb '[OP_1 OP_2 OP_ADD]'  
btcdeb -- type `./btcdeb -h` for start up options  
valid script  
3 op script loaded. type `help` for usage information  
script | stack  
-----+-----  
1      |  
2      |  
OP_ADD |  
#0000 1  
btcdeb> step
```

More OP codes: Arithmetic and Boolean

OP_1ADD (0x8b)	— Increment by one
OP_1SUB (0x8c)	— Decrement by one
OP_NEGATE (0x8f)	— Flip the sign of the number
OP_ABS (0x90)	— Make the number positive
OP_NOT (0x91)	— Flips 1 and 0, else 0
OP_ADD (0x93)	— Add two numbers
OP_SUB (0x94)	— Subtract two numbers
OP_MIN (0xa3)	— Return the smaller of two numbers
OP_MAX (0xa4)	— Return the larger of two numbers
OP_BOOLAND (0x9a)	— 1 if both numbers are not 0, else 0
OP_BOOLOR (0x9b)	— 1 if either number is not 0, else 0

More OP codes: Equality and Stack

OP_NUMEQUAL (0x9c) — 1 if both numbers are equal, else 0
OP_LESSTHAN (0x9f) — 1 if first number is less than second, else 0
OP_GREATERTHAN (0xa0) — 1 if first number is greater than second, else 0
OP_LESSTHANOEQUAL (0xa1) — 1 if first number is less than or equal to second, else 0
OP_GREATERTHANOEQUAL (0xa2) — 1 if first number is greater than or equal to second, else 0
OP_WITHIN (0xa5) — 1 if a number is in the range of two other numbers

OP_DEPTH (0x74) — Pushes the size of the stack
OP_PICK (0x79) — Duplicates the nth stack item as the top of the stack
OP_ROLL (0x7a) — Moves the nth stack item to the top of the stack
OP_SWAP (0x7c) — Swaps the top two stack items

More OP codes: Cryptogr. and Conditional

OP_RIPEMD160 (0xa6)	— RIPEMD-160
OP_SHA1 (0xa7)	— SHA-1
OP_SHA256 (0xa8)	— SHA-256
OP_HASH160 (0xa9)	— SHA-256 + RIPEMD-160
OP_HASH256 (0xaa)	— SHA-256 + SHA-256
OP_CHECKSIG (0xac)	— Check a signature
OP_CHECKMULTISIG (0xae)	— Check a m-of-n multisig
OP_IF (0x63)	— If top stack item true execute block (up to OP_ELSE, if there)
OP_ELSE (0x67)	— If OP_IF top stack item is false executes OP_ELSE block
OP_ENDIF (0x68)	— Ends a if/else block



P2SH

P2SH (1)

- Redeem (locking) script is provided by the redeemer! (Why?)
- Multisig, old style
 - scenario: company with 2-of-3 multisig locking script:

```
2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG
```

- To receive money
 - company needs to send the address and the redeem script to customers
 - impractical
 - privacy implications
 - inefficient storage

P2SH (2)

A P2SH moves the responsibility for supplying the conditions to redeem a transaction (locking script) from the sender of the funds to the redeemer (receiver).

- The locking script of such a transaction is quite simple:
`OP_HASH160 [20-byte-hash-value] OP_EQUAL`
- The 20-byte hash is the hash of the redeem script:

```
RIPEND-160( SHA-256( 2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG ) )
```

- Using this hash we create a Bitcoin address (same process but instead of `OP_HASH160(pubkey)` we use `OP_HASH160(redeem script)`) using the version prefix of 0x05 that creates addresses that start with 3.
- We then disseminate only this address to the company's customers to send the funds.

P2SH (3)

When the company needs to spend the funds it would send the following unlocking script:

```
<Director's signature> <CFO's signature> <2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG >
```

Validation occurs in 2 steps. First we confirm that the redeem script equals the hash in the locking script:

```
<Director's signature> <CFO's signature> <2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG> OP_HASH160  
[20-byte-hash-value] OP_EQUAL
```

And then just validating the unlocking script:

```
<Director's signature> <CFO's signature> 2 <Director's Public Key> <CFO's Public Key> <COO's Public Key> 3 CHECKMULTISIG
```

Transaction Types

Any transaction not of the following types:

- P2PK (TX_PUBKEY)
- P2PKH (TX_PUBKEYHASH)
- P2SH (TX_SCRIPTHASH)
- P2WPKH (TX_WITNESS_V0_KEYHASH)
- P2WSH (TX_WITNESS_V0_SCRIPTHASH)
- OP_RETURN (TX_NULL_DATA)
- Multisignature (TX_MULTISIG)
- Non-standard (TX_NONSTANDARD)

Non-standard transactions are rejected (but not invalid) and not relayed by nodes.



Creating Transactions

Using bitcoin-cli -- Node's API, high-level

```
./bitcoin-cli listunspent 0
[
  {
    "txid": "b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",
    "vout": 0,
    "address": "n1jnmQCyt9DHR3BYKzdbmXWM8M5UvH9nMW",
    "account": "",
    "scriptPubKey": "76a914ddcf9faf5625d6a96790710bbcef98af9a8719e388ac",
    "amount": 1.30000000,
    "confirmations": 0,
    "spendable": true,
    "solvable": true
  }
  ...
]
```

```
./bitcoin-cli sendtoaddress mnB6gSoVfUAPu6MhKkAfgsjPfbWmEEemFr3 0.1
```

Using bitcoin-cli - Node's API, low-level (1)

```
./bitcoin-cli createrawtransaction ''  
> [  
>   {  
>     "txid":  
"b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",  
>     "vout": 0  
>   }  
> ]  
> '' ''  
> {  
>   "mqazutWCSnuYqEpLBznke2ooGimyCtwCh8": 0.2  
> }''  
01000000014039182702194345ac14bcc8ea624a72710b6279c1d5a43de8a972344d46b7b3000000  
0000fffffffff01002d3101000000001976a9146e751b60fcb566418c6b9f68bfa51438aefbe09488  
ac00000000
```

Using bitcoin-cli - Node's API, low-level (2)

```
./bitcoin-cli decoderawtransaction
01000000014039182702194345ac14bcc8ea624a72710b6279c1d5a43de8a972344d46b7b3000000000000ffffffffff01002d
31010000000001976a9146e751b60fcb566418c6b9f68bfa51438aefbe09488ac00000000
{
  "txid": "a7b54334096108e8f69ecfa19263cfbf2f12210165ef5fc2e98ef8e4e466392e",
  "hash": "a7b54334096108e8f69ecfa19263cfbf2f12210165ef5fc2e98ef8e4e466392e",
  "size": 85,
  "vsize": 85,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "b3b7464d3472a9e83da4d5c179620b71724a62eac8bc14ac4543190227183940",
      "vout": 0,
      "scriptSig": {
        "asm": "",
        "hex": ""
      },
      "sequence": 4294967295
    }
  ],
}
```

Using bitcoin-cli - Node's API, low-level (3)

```
"vout": [  
  {  
    "value": 0.20000000,  
    "n": 0,  
    "scriptPubKey": {  
      "asm": "OP_DUP OP_HASH160 6e751b60fcb566418c6b9f68bfa51438aefbe094 OP_EQUALVERIFY OP_CHECKSIG",  
      "hex": "76a9146e751b60fcb566418c6b9f68bfa51438aefbe09488ac",  
      "reqSigs": 1,  
      "type": "pubkeyhash",  
      "addresses": [  
        "mqazutWCSnuYqEpLBznke2ooGimyCtwCh8"  
      ]  
    }  
  }  
]
```

Using bitcoin-cli - Node's API, low-level (4)

```
./bitcoin-cli signrawtransaction
01000000014039182702194345ac14bcc8ea624a72710b6279c1d5a43de8a972344d46b7b3000000000f
ffffffff01002d3101000000001976a9146e751b60fcb566418c6b9f68bfa51438aefbe09488ac00000000
{
  "hex":
"01000000014039182702194345ac14bcc8ea624a72710b6279c1d5a43de8a972344d46b7b3000000006a
4730440220404082ecae0b088e07647a5a4eb5c71626e001cbca9353bb6f7e6b212f0f95d002202cdadf6
4f31b11e1901134abe7917d74105953aa983db099504891696277b86d01210306a6ae64fbb424a81260a6
c47f3cb52eec39c4b40ded8b05e150458b95ea6465ffffffff01002d3101000000001976a9146e751b60f
cb566418c6b9f68bfa51438aefbe09488ac00000000",
  "complete": true
}
```

```
./bitcoin-cli decoderawtransaction 01000000014039..8ac00000000
```

```
...
```

```
./bitcoin-cli sendrawtransaction 01000000014039..8ac00000000
```

```
error code: -26, error message:, 256: absurdly-high-fee
```

Using HTTP JSON-RPC (1)

- Simple protocol for communication
 - defines format of request/respond messages
 - transport agnostic
- Bitcoin.conf
 - Only local connections allowed
 - rpcallowip

```
rpcuser=kostas  
rpcpassword=too_long_to_guess
```

Using HTTP JSON-RPC (2)

```
$ curl --user kostas --data-binary '{"jsonrpc": "1.0", "id": "curltest",  
"method": "getinfo", "params": [] }' -H 'content-type: text/plain;'  
http://127.0.0.1:18332/  
Enter host password for user 'kostas':
```

```
{  
  "result":  
    {  
      "version": 130100,  
      "protocolversion": 130000,  
      "balance": 2.27500000,  
      ...  
    }  
  "error": null,  
  "id": "curltest"  
}
```

Using a library

Using `python-bitcoinrpc`. Install with `pip` and try it out.

```
from bitcoinrpc.authproxy import AuthServiceProxy, JSONRPCException

# user and pw are rpcuser and rpcpassword respectively
user = "kostas"
pw = "too_long_to_guess"      # bad practice !!
rpc_connection = AuthServiceProxy("http://%s:%s@127.0.0.1:18332"%(user, pw))
block_count = rpc_connection.getblockcount()
print(block_count)
```

All API calls can be used, including the ones to create a transaction with either `sendtoaddress` or `createrawtransaction + signrawtransaction + sendrawtransaction`.

Using another library (1)

Using `python-bitcoinlib`. Install with `pip` and try it out.

```
import bitcoin.rpc
from bitcoin import SelectParams
from bitcoin.core import COIN, b2lx
from bitcoin.wallet import CBitcoinAddress

SelectParams('testnet')

rpc = bitcoin.rpc.Proxy()
addr = CBitcoinAddress('mwtaAhm3Fdjnc525kMENrpP7zsqE8VvWdZ')

txid = rpc.sendtoaddress(addr, 0.001 * COIN)
print(b2lx(txid))
```

Using another library (2)

Using the same library we can construct the transaction from scratch! Please consult this [example](#) for a P2PKH transaction.

- What does this example do?
- What is need to complete the transaction?
- Can you identify a problem with that example?

And [here](#) is another example for a P2SH transaction that contains the obsolete P2PK transaction in the redeem script (`<pubKey> OP_CHECKSIG`).



Questions ?