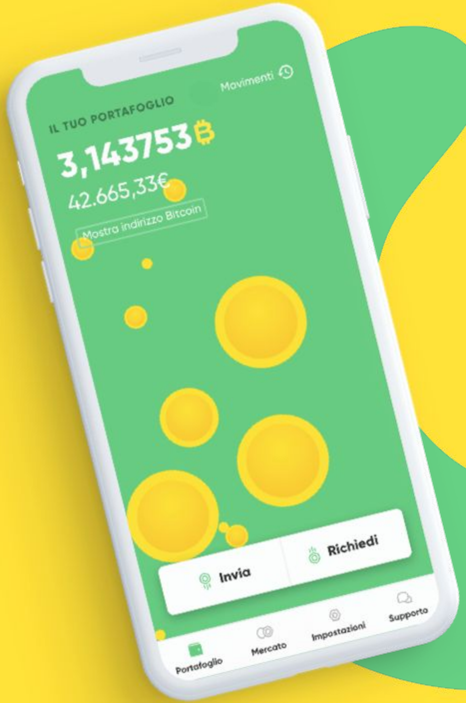
Bitcoin 101 with Python.

PyMI - Python Milano

July 11, 2018



Why

01

Scattered / stale documentation on Bitcoin

02

Steep learning curve, especially for beginners

03

Missing consistent reference libraries

What

- **Bitcoin basic concepts**
  - Blockchain
  - Keys (single sig) and address
  - Transaction
  - Hierarchical keys
- **Local environment configuration**
- **Examples in Python using Pybitcointools**

https://github.com/conio/pybitcointools

What is <u>not</u>

- A full dive into Bitcoin internals
- A Bitcoin trading guide
- A walkthrough to create a production-ready Bitcoin app
- A business-oriented approach to the adoption of Bitcoin

Bitcoin basic concepts.

What is Bitcoin

- It is a cryptocurrency created in 2009 by Satoshi Nakamoto
- It defines a completely decentralized protocol
- It is based on a trustless, ~~distributed~~ decentralized and unmodifiable database known as BlockChain

What is a cryptocurrency

- **A digital currency:**
  no physical coins
- **The ownership is based on a cryptographic secret**

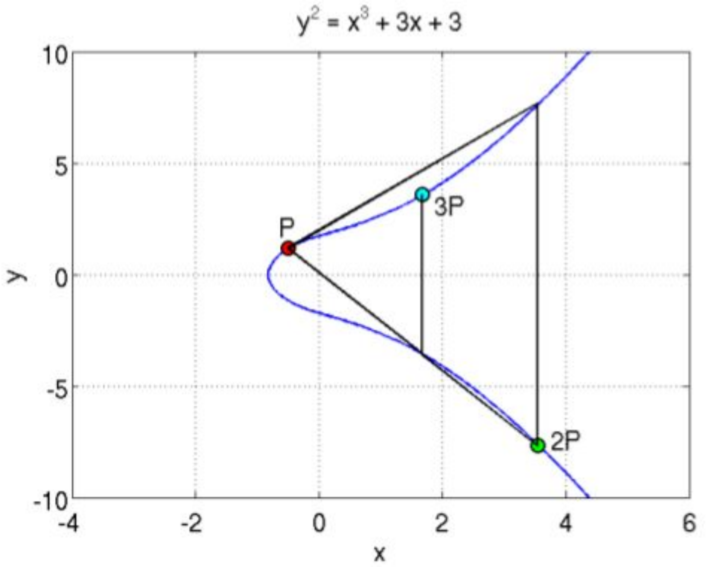
- **For Bitcoin**
- it is based on ECDSA/secp256k1
- The owner of Bitcoins know his ECDSA private key
- Given the corresponding public key, we have a bitcoin address, as:

```
BASE58_CHECK(
    <VBYTE> || RIPEMD160(SHA256(pubkey))
)
```

VBYTE=0x00 for mainnet, 0x6f for testnet

What is ECDSA

- **ECDSA stands for Elliptic Curve Digital Signature Algorithm**
- **It is based on the elliptic curve**
  $y^2=x^3+ax+b \ (mod\ p)$, with x, y, a, b ∈ I



What is ECDSA

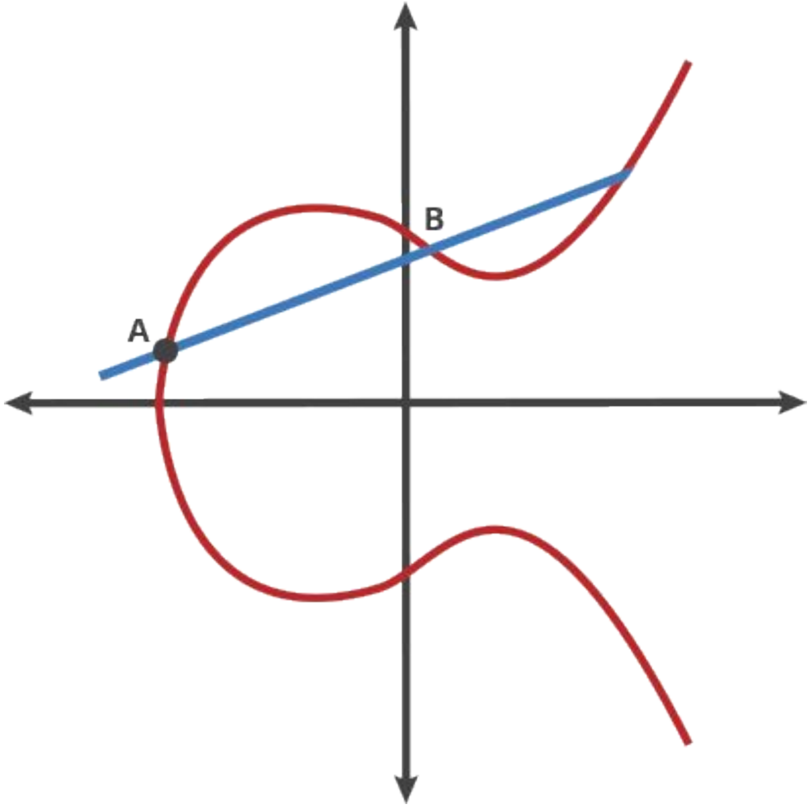
- **A private key**
  is an integer number dA
- **A public key Qa**
  is a point in the elliptic curve
- **Given G as base point for the curve**
  $Qa = dA \times G = (Qa_x, Qa_y)$ (point multiplication)

- **Signature of a document z:**
  Rx=x coordinate of the point k x G, with k random int
  $S = k^{-1} * (z + dA * R_x) \bmod p$

Verification:

Given $P = S^{-1} * z \times G + S^{-1} * R_x \times Qa$, stands that:
P == Rx

What is ECDSA



What is ECDSA

- **The SEC1v2 public key representation is used to generate the Bitcoin address**
- **Given a private key, I get 2 different representations of the same public key**
- **As an alternative, the public key can be stored as:**
  <02|03>R, with prefix 02 if S is even, otherwise 03
- **Given the two public key representations, I get two different bitcoin addresses**

Python lab: Keys.

Decentralization

- **Bitcoin**
  is based on
  a decentralized protocol

- **Each node**
  contributes to the *consensus algorithm*, which determines the validity of each block

- **Some nodes**
  run the role of *miners*, who are in charge of modifying the blockchain

- **The *proof of work***
  in a healthy and balanced network ensures the correctness of the blockchain data

- **Each node**
  runs a *bitcoind* daemon
  (or a compliant software)

- **Bitcoind**
  provides an RPC interface (*bitcoin-cli*) to interact with the Bitcoin network

Decentralization

Some useful RPC CLI commands:

- `bitcoin-cli getinfo # gets the blockchain height and other general stuff`
- `bitcoin-cli getrawtransaction <tx_hash> 1 # gets JSON tx structure`
- `bitcoin-cli decoderawtransaction <raw_tx> # decodes tx into JSON`
- `bitcoin-cli generate <num_blocks> # generates <num_blocks> blocks`
- `bitcoin-cli getrawmempool # gets the Mempool contents`
- `bitcoin-cli sendtoaddress <btc_address> <amount> # send <amount> BTC to <btc_address>`
- `bitcoin-cli help # list of all possible commands`

Decentralization

- **Bitcoin provides 3 networks:**
  - main
  - test
  - regtest
- **Each network**
  has its VBYTE code
- **Apart from the VBYTE code**
  the software behaves the same as in main network
- **Regtest**
  we will use Regtest to do some experiments without burning real coins

What is the proof of work

- **Blocks**
  must contain only valid data (e.g. must not spend money twice)
- **Each block**
  Since the Blockchain is trustless, it is necessary to guarantee the validity of each block without trusting miners
- **Without additional forces, anyone could**
  - generate invalid blocks
  - attempt to double spend
  - create denial of services
  - benefit from coinbase
- **Block hashes**
  must comply with the *difficulty* of the network
- **Complying with the difficulty**
  is a *hard* task and requires about 10 minutes of bruteforce computation

Lab: create your own “network”.

What is the BlockChain

- It is a decentralized database
- It is accessible to anyone
- Any attempt to modify it must follow the *consensus* rules
- To some extent, it is immutable
- It's trustless
- It is made of blocks, which are *mined* by *miners*
- It is *literally* an ordered sequence of immutable blocks

What is the BlockChain

- **Each block contains transactions**
- **The first transaction is *always* the coinbase**
- **Each block has a unique id that depends on**:
  - the previous block id
  - the transaction ids it contains
  - its timestamp
- **It is *hard* to create a block,**
  but it is relatively easy to verify it
- **Violation**
  Any violation of the rules by *miners* causes the rejection of the block and a loss of money for the miner in terms of waste of time and energy

What is the BlockChain

- **Transaction**
  Are stored in the *mempool* before being *included* in a block by *miners*

- When a transaction is included its *number of confirmations* is 1

- For each next generated block, the number of confirmations is increased by 1

When the number of confirmations is 6, the transaction is *conventionally* confirmed

What is the BlockChain

- **Fork**
  Sometimes the BlockChain *forks*, that is, the network experiences a brain split on the right sequence of blocks

- Each miner has a vested interest in generating the block in right branch of the *fork*

- Each miner appends blocks in the longest chain of the fork

- As a result, the whole network actively works to consistently promote the longest fork

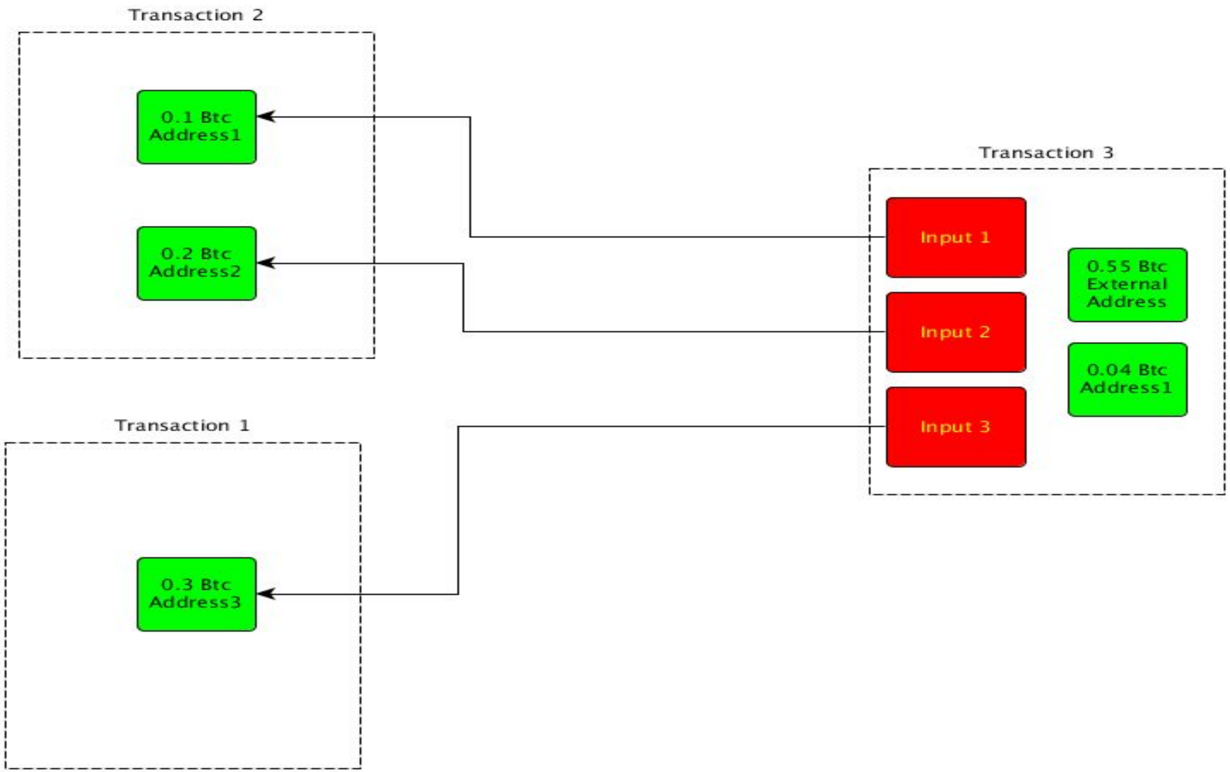
When a transaction is confirmed, it is highly unlikely that it may be engaged in a fork

What is a transaction

- **It is a transfer of the ownership of funds**
- **Transactions are made of *outputs* and *inputs***
- **Output**
  Each output specifies the destination of the funds
- **Input**
  Each input is a reference to previous outputs that will pay for the transaction

There may be a *change* output to return the extra Bitcoins to his owner

What is a transaction



What is a transaction

- **Transferred funds can only be unlocked by private keys**
- **Each input must have the necessary cryptographic info to refer a specific output**

- **When is valid**
  A transaction is valid if it does not refer to already spent outputs and if all the crypto info are valid
- **Input and output**
  The difference between coins referred by inputs and coins specified in outputs (change output included) is the *mining fee*

The more you pay in terms of mining fees, the higher the probability to be included in the next block

What is a transaction

- **Inputs have a script field (ScriptSig)**
- **ScriptSig contains**
  - The signature of the unsigned transaction
  - The public key of key private used to sign the transaction

- **Outputs have a script field (ScriptPubKey)**
- **ScriptPubKey contains**
  - The hash of the public key that is expected to be found in the ScriptSig
  - Some opcodes that specify how to validate the signature

What is a transaction

- ScriptSig and ScriptPubKey of the referred output are run to verify the encryption rules
- The execution of the resulting script must return True and leave the execution stack empty

There are many scripts schemas, but for semplicity we will cover just *single sig with SIGHASH_ALL*

What is a transaction

A. **Example**:
   a. ScriptSig: <signature> <public key>
   b. ScriptPubKey: DUP HASH_160 <expected hash> EQUAL_VERIFY CHECKSIG

B. **Execution stack**
   a. [signature, public key]
   b. [signature, public key, DUP]
   c. [signature, public key, public key]
   d. [signature, public key, public key, HASH_160]
   e. [signature, public key, hash]
   f. [signature, public key, hash, expected hash]
   g. [signature, public key, hash, expected hash, EQUAL_VERIFY]
   h. [signature, public key]
   i. [signature, public key, CHECKSIG]
   j. []

What is a transaction

Spending transaction

```
"txid": "e9a66845e05d5abc0ad04ec80f774a7e585c6e8db975962d069a522137b80c1d",
"vin": [
            {
                "txid":
"f4515fed3dc4a19b90a317b9840c243bac26114cf637522373a7d486b372600b",
                "vout": 0,
                "scriptSig": {
                    "asm": "30460221<32 bytes>0221<32 bytes>[ALL]
04a7135bfe824c97ecc01e...063ce6af4a4ea14fbb"},
            }
],
"vout": [ ...]
```

What is a transaction

Spent transaction


```
"vout": [

                {
                     "value": 0.01000000,
                     "n": 0,
                     "scriptPubKey": {
                          "asm": "OP_DUP OP_HASH160
c4eb47ecfdcf609a1848ee79acc2fa49d3caad70 OP_EQUALVERIFY OP_CHECKSIG",
                          "hex": "76a914c4eb47ecfdcf609a1848ee79acc2fa49d3caad7088ac",
                          "reqSigs": 1,
                          "type": "pubkeyhash",
                          "addresses": [
                               "1JxDJCyWNakZ5kECKdCU9Zka6mh34mZ7B2"
                          ]
                     }
                }
           ],
```

Lab: create transaction.

Hierarchical keys

- **Using always the same key to receive and spend coins**
  - may not be cryptographically safe
  - may cause a disclosure of your privacy
  - in case of server-side wallets, you may want to keep users cryptographically isolated

- **Random keys**
  Having multiple random keys may be unmanageable in terms of security and usability

- **Master key**
  It may be easier to generate a master key from something that is easy to remember and *derive* all the subkeys from it

Hierarchical keys

- **BIP32** (**https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki**) defines a standard way to deterministically generate *extended* keys from a master key

- **Extended keys** (k, c) are made of key (k) + metadata (such as the *chain code*)
  - Es. xprv9s21ZrQH143K3QTDL4LXw2F7HEK3wJUD2nW2nRk4stbPy6cq3jPPqjiChkVvvNKmPGJxWUtg6LnF5kejMRNNU3TGtRBeJgk33yuGBxrMPHi
    - 0488ade4: prefix
    - 00: depth
    - 00000000: parent fingerprint
    - 00000000: child number
    - 873dff81c02f525623fd1fe5167eac3a55a049de3d314bb42ee227ffed37d508: chain code
    - 00e8f32e723decf4051aefac8e2c93c9c5b214313817cdb01a1494b917c8436b35: key
    - e77e9d71: checksum

Hierarchical keys

- **Given an extended key and a *numeric path*, it is possible to get a new extended key**
- **Given a master key, it is possible to create a tree made of extended keys;** each connection is an element of the numeric path
- **Numeric path**
  The numeric path is made of the path between the master key and the selected extended key
- **Extended keys may be public or private**

The master key, which is the root of the whole tree, is an extended key

Hierarchical keys

Key1 path: [1]

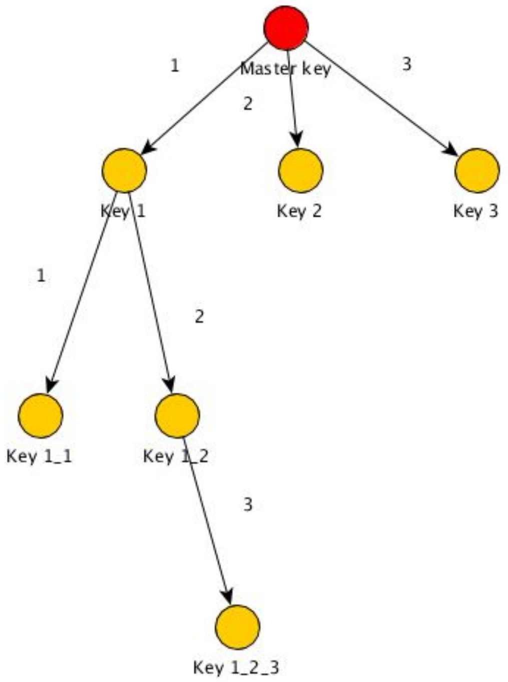
Key2 path: [2]

Key3 path: [3]

Key1_1 path: [1, 1]

Key1_2 path: [1, 2]

Key1_2_3 path: [1, 2, 3]



Hierarchical keys

Key derivation may be non-hardened or hardened

- **Non-hardened key (path element $i < 2^{31}$):**
  - I_{priv} = HMAC-SHA512(Key = c_{par}, Data = serP(point(kp_{priv})) || ser32(i))
  - kc_{priv} = I[:32] + kp_{priv}
  - I_{pub} = HMAC-SHA512(Key = c_{par}, Data = serP(kp_{pub}) || ser32(i))
  - kc_{pub} = point(I[:32]) + kp_{pub}
  - Follows that kc_{pub} is the public key of kc_{priv}
  - You can generate public keys without knowing the private keys

---

- **Hardened key ($i >= 2^{31}$):**
  - I = HMAC-SHA512(Key = c_{par}, Data = 0x00 || ser256(kp_{priv}) || ser32(i))
  - kc_{priv} = I[:32] + kp_{priv}
  - Public key derivation is not possible

Hierarchical keys

The master key is the most important element in the hierarchy

- It is important to generate it in a cryptographically safe way
- It would be also nice to have a way to simply remember the master key
- We might generate a key from a mnemonic passphrase using an algorithm that makes bruteforce unfeasible

Hierarchical keys

BIP39

defines how to generate the passphrase and derive the master key from it

- The mnemonic is generated from a defined dictionary
- The master key is generated as PBKDF2(seed(mnemonic), salt=a passphrase, pseudo_random_func=HMAC_SHA512)

(https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki)

BIP44

defines a standard for the numeric path as follows:

- `m / purpose' / coin_type' / account' / change / address_index`

(https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki)

Lab: deterministic keys

References

Mastering bitcoin: https://goo.gl/ouACGx

Bitcoind software: https://bitcoin.org/it/scarica

BIP specs: https://github.com/bitcoin/bips

Pybitcointools: https://github.com/conio/pybitcointools

SEC1v2: http://www.secg.org/sec1-v2.pdf

Materials: https://github.com/fcracker79/python_bitcoin_101

Q&A

Contacts

Mirko Bonasorte

Email: mirko@conio.com

Website: www.mirko.io

Conio Inc.

Website: www.conio.com

github.com/conio

github.com/fcracker79

Thank you.