XCS251 Assignment 2: Perform Bitcoin transactions using python-bitcoinlib

---

Due Sunday, September 25 at 11:59pm PT.

Guidelines

1. If you have a question about this assignment, we encourage you to post your question on slack channel.
2. Familiarize yourself with the collaboration and honor code policy before starting work.

Submission Instructions

You should submit a PDF with your solutions online in Gradescope. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Download and install Tex Live or try Overleaf.
- Submit the compiled PDF.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on assignment in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the assignment the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Introduction

In this assignment, you will gain experience creating transactions using the Bitcoin and BlockCypher testnet blockchains and Bitcoin Script. This assignment consists of four questions, and the starter code we provide uses python-bitcoinlib, a free, low-level Python 3 library for manipulating Bitcoin transactions. Details on the background of bitcoin transaction and testnet, the instructions on getting started with the started code, and setup steps to interact with the Bitcoin testnet will be provided in the handout. By the end of this assignment, you should have a good understanding of how Bitcoin transactions work.

1 Project Background

1.1 Anatomy of a Bitcoin Transaction

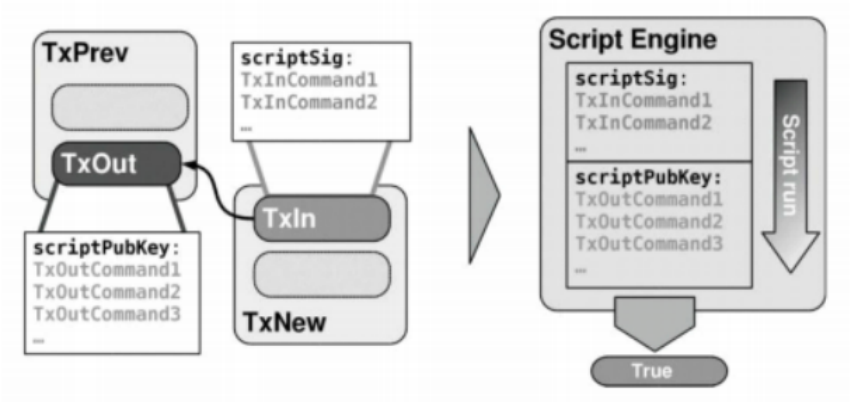

Figure 1: Each `TxIn` references the `TxOut` of a previous transaction, and a `TxIn` is only valid if its scriptSig outputs True when prepended to the `TxOut`'s scriptPubKey.

Bitcoin transactions are fundamentally a list of outputs, each of which is associated with an amount of bitcoin that is "locked" with a puzzle in the form of a program called a scriptPubKey (also sometimes called a "smart contract"), and a list of inputs, each of which references an output from the list of outputs and includes the "answer" to that output's puzzle in the form of a program called a scriptSig. Validating a scriptSig consists of appending the associated scriptPubKey to it, running the combined script and ensuring that it outputs True.

$$\text{run}(\textbf{scriptSig} \mid\mid \textbf{scriptPK}) = \begin{cases} \text{True} & \text{valid scriptSig, TxIn spend TxOut} \\ \text{else} & \text{invalid scriptSig, TxIn cannot spend TxOut} \end{cases}$$

Most transactions are "PayToPublicKeyHash" or "P2PKH" transactions, where the scriptSig is a list of the recipient's public key and signature, and the scriptPubKey performs cryptographic checks on those values to ensure that the public key hashes to the recipient's bitcoin address and the signature is valid.

Each transaction input is referred to as a `TxIn`, and each transaction output is referred to as a `TxOut`. The situation for a transaction with a single input and single output is summarized by Figure 1 above.

The sum of the bitcoin in the unspent outputs to a transaction must not exceed the sum of the inputs for the transaction to be valid. The difference between the total input and total output is implicitly taken to be a transaction fee, as a miner can modify a received transaction and add an output to their address to make up the difference before including it in a block.

$$\sum \text{TxIn} = \sum \text{TxOut} + \text{Tx}_{\text{fee}}$$

For the first 3 questions in this project, the transactions you create will consume one input and create one PayToPublicKeyHash output that sends an amount of bitcoin back to the testnet faucet. The 4th question will carry out a swap of coins between two entities, Alice and Bob. For these exercises, you will want to take the fee into account when specifying how much to send and subtract a bit from the amount in the output you're sending, say 0.001 BTC (this is just to be safe, you can probably include a fee as low as 0.00001 BTC if your funds are running low). If you do not include a fee, it is likely that your transaction will never be added to the blockchain. Since BlockCypher (see Section 1.3) will delete transactions that remain unconfirmed after a day or two, it is very important that you include a fee to make sure that your transactions are eventually confirmed.

1.2 Script Opcodes

Your code will use the Bitcoin stack machine's opcodes, which are documented on the Bitcoin wiki [1]. When composing programs for your transactions' scriptPubKeys and scriptSigs you may specify opcodes by using their names verbatim. For example, below is an example of a function that returns a scriptPubKey that cannot be spent, but rather acts as storage space for an arbitrary piece of data that someone may want to save to the blockchain using the `OP_RETURN` opcode.

```
def save_message_scriptPubKey(message):
    return [OP_RETURN,
            message]
```

Examples of some opcodes that you will likely be making use of include `OP_DUP`, `OP_CHECKSIG`, `OP_EQUALVERIFY`, and `OP_CHECKMULTISIG`, but you will end up using additional ones as well.

1.3 Overview of Testnets

Rather than having you download the entire testnet blockchain and run a bitcoin client on your machine, we will be making use of an online block explorer to upload and view transactions. The one that we will be using is called BlockCypher, which features a nice web interface as well as an API for submitting raw transactions that the starter code uses to broadcast the transactions you create for the exercises. After completing and running the code for each exercise, BlockCypher will return a JSON representation of your newly created transaction, which will be printed to your terminal. An example transaction object along with the meaning of each field can be found at BlockCypher's developer API documentation at https://www.blockcypher.com/dev/bitcoin/#tx. Of particular interest for the purposes of this project will be the `hash`, `inputs`, and `outputs` fields. Note that you will be using two different test networks ("testnets") for this project: the Bitcoin testnet (the current version is Testnet3) for questions 1-4 and the BlockCypher testnet for question 4. These will be useful in testing your code. As part of these exercises, you will request coins to some addresses (more details below).

2 Getting Started

1. Download the starter code, navigate to the directory, create and activate python virtual environment and run `pip install -r requirements.txt` to intall the required dependencies. For this project, ensure that you are using Python 3. You can type `python --version` to find out if you are using Python 3 under `python` command. You can type `pip --version` to find out if you are using Python 3 under `pip` command. To create, activate, and deactivate a Python virtual environment, check https://packaging.python.org/en/latest/tutorials/installing-packages/#creating-virtual-environments. If you are not using a Python virtual environment, you must do two things differently. First, use `pip3` instead of `pip` to install packages to Python 3. Second, use the `python3` command to run scripts instead of `python` to run with the Python 3 interpreter.

2. Make sure you understand the structure of Bitcoin transactions and read the references in the Recommended Reading section 7 if you would like more information.

3. Read over the starter code. Recommend the use of an IDE for simplicity. Visual Studio Code or PyCharm are excellent. Here is a summary of what each of the files contain:

   `lib/keygen.py`:

   You will run this script to generate new private keys and corresponding addresses for the Bitcoin Testnet. Questions 1-3 will solely use these private keys, while question

4 will also require you to use an alternative method to generate Block Cypher Testnet keys. **You are not expected to modify this file.**

`lib/split_test_coins.py`:

You will run this script to split your coins across multiple unspent transaction outputs (UTXOs). You will have to edit this file to input details about which transaction output you are splitting, the UTXO index, etc.

`lib/config.py`:

You will modify this file to include the private keys for your users. Note that `my_private_key`, `alice_secret_key_BTC` and `bob_secret_key_BTC` will be generated using the `lib/keygen.py` file. You will make web requests to generate `alice_secret_key_BCY` and `bob_secret_key_BCY`. **There are comments in `config.py` and instructions during setup for how to do this**.

`lib/utils.py`:

Contains various util methods. **You are not expected to modify this file.**

`Q1.py, Q2a.y, Q2b.py, Q3a.py, Q3b.py, Q4.py`:

You will have to modify the various scriptSig and scriptPubKey methods, as well as fill the transaction parameters. Note that for question 3, you will have to generate additional private and public keys for customers using the `lib/keygen.py` file.

`alice.py, bob.py`:

Creates and submits transactions for Q4 on behalf of Alice and Bob. **You are not expected to modify these files.**

`swap.py`

Contains the logic to carry out the atomic swap. **You are not expected to modify this file.**

`docs/transactions.py`

You are expected to fill this file with the transaction ids generated for questions 1-3.

`docs/Q4design_doc.txt`

You are expected to fill this design doc to explain your solution to Q4.

4. Be sure to start early on this project, as block confirmation times can vary depending on how busy the network is!

3 Setup

1. Open `lib/config.py` and read the file. Note that there are several users that you will need to generate private keys and addresses for.

2. First we are going to generate key pairs for you, Alice, and Bob on the Bitcoin Testnet. Run `lib/keygen.py` to generate private keys for `my_private_key`, `alice_secret_key_BTC` and `bob_secret_key_BTC`, and record these keys in `lib/config.py`. Note that Alice and Bob's keys will only come into play for question 4. Please make sure to create different keys for Alice and Bob, you wouldn't want them to be able to forge each others' transactions! You may want to record each generated private key and address in a separate file and note the corresponding user for easy reference.

3. Next, we want to get some test coins for `my_private_key` and `alice_secret_key_BTC`. To do so:

   (a) Go to the Bitcoin Testnet faucet (https://testnet-faucet.mempool.co/) and paste in the corresponding addresses of the users. Note that faucets will often rate-limit requests for coins based on Bitcoin address and IP address, so try not to lose your test Bitcoin too often. It is recommended that you use the address associated with `my_private_key` with the first faucet listed above since that faucet gives more coins and you will be performing more exercises with that address. Note that the faucet limits requests by the same IP address to max 0.002 per hour and 0.002 per request. Therefore, use lesser amount for each address to avoid max out.

   (b) You may want to record the transaction hash, i.e., TxID, in the same file that you record the generated private key and address for each user. An example screenshot is provided in Figure 2 below.

   Viewing the transaction in a block explorer (e.g. https://live.blockcypher.com/) will also let you know which output of the transaction corresponds to your address, and you will need this `utxo_index` for the next step as well. For example, if you search the address that you just requested a coin in Bitcoin Testnet, you can see the transaction with the TxID in Figure 3 below.

   Requested value 0.001 is the first output, i.e., `utxo_index` 0. If it is the second output, then it will be `utxo_index` 1. You will also see "0/6 confirmations". You will want to wait for fully verified (at least 6/6 confirmations) to split the coin as described in step 6.

   If the faucet doesn't give you a transaction hash, you can also paste the user address into the block explorer and find the transaction that way. It is best to navigate the blocks using the user address if the transaction hash does not work.

Figure 2: Screenshot of TxID.

4. Next, we are going to create generate key pairs for Alice and Bob on the BlockCypher testnet.

   (a) Sign up for an account with Blockcypher to get an API token here: https://accounts.blockcypher.com/
   You need a valid email to confirm the account. Please save the new token in a text file for future reference.

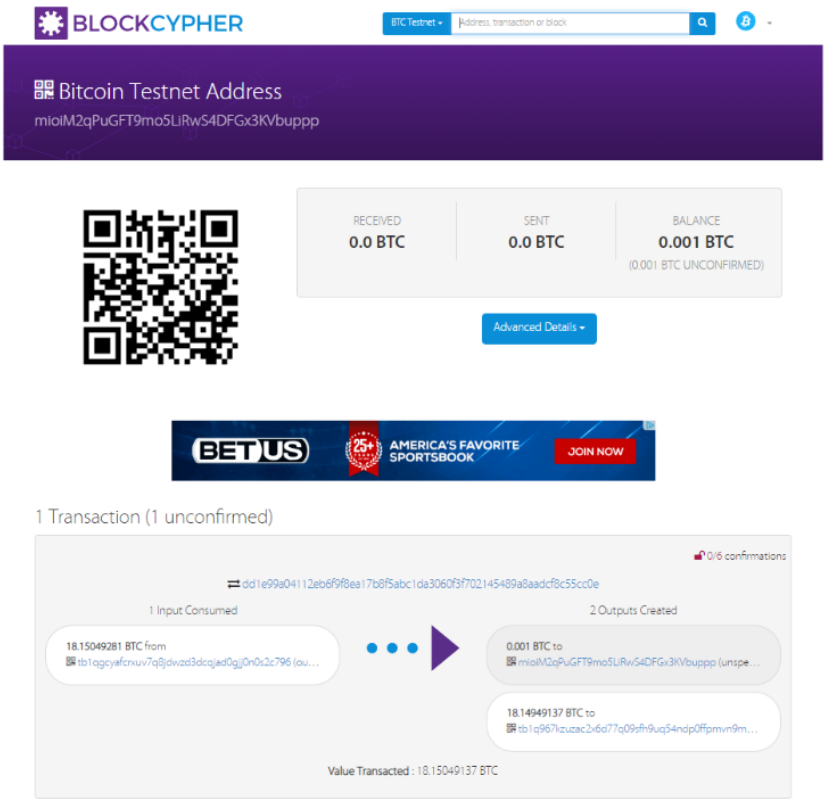

Figure 3: Screenshot of Transaction.

(b) Create BCY testnet keys for Alice and Bob and place into `lib/config.py`.

```
curl -X POST 'https://api.blockcypher.com/v1/bcy/test/addrs?token=YOURTOKEN'
```

Note, if you copy this command directly into your terminal from this handout, you'll likely need to delete and retype the ' for the command to work. Also note that for Windows OS, you will want to use double quote rather than single quote like below

```
curl -X POST "https://api.blockcypher.com/v1/bcy/test/addrs?token=YOURTOKEN"
```

Remember to record the results in a separate file like before and note the corresponding user for easy reference

5. Give Bob's address bitcoin on the Blockcypher testnet (BCY) and record the transaction hash.

```
curl -d '{"address": "BOBS_BCY_ADDRESS", "amount": 100000}' \
'https://api.blockcypher.com/v1/bcy/test/faucet?token=YOURTOKEN'
```

Note, if you copy this command directly into your terminal from this handout, you'll likely need to delete and retype the `'{` and the `}'`, delete the `\`, retype the `'` around the url, and condense the command into one line for it to work. It is preferred to type the command to avoid reformatting issues. Also note that for Windows OS, you will want to use double quote rather than single quote.

6. The faucets will give you, Alice, and Bob one spendable output per person, but we would like to have multiple outputs to spend in case we accidentally lock some with invalid scripts. Edit the parameters at the bottom of `split_test_coins.py`, where `txid_to_spend` is the transaction hash from the faucet to your address, `utxo_index` is 0 if your output was first in the faucet transaction and 1 if it was second as described before, and `n` is the number of outputs you want your test coins split evenly into, and run the program with `python split_test_coins.py`. A perfect run through of questions 1-3 would require `n` = 3 for your address, one for each exercise, but if you anticipate accidentally locking an output due to a faulty script a couple times per exercise then you might want to set `n` to something higher like 8 so that you don't have to wait to access the faucet again or have to try with a different Bitcoin address. **If `split_test_coins.py` was successful, you should get back some information about the transaction.** Record the entire result with the transaction hash in a separate file like before, as each exercise will be spending an output from this transaction and will refer to it using this hash.

   Note: The faucet transaction would need to be fully verified (at least 6/6 confirmations) before you can split the coins you received. Waiting times will vary based on how busy the network is.

7. You should also split Alice's and Bob's coins into multiple outputs just to be safe. Record the entire result in a separate file like before. Note that each time you switch between the Bitcoin and BlockCypher tesntets, you should adjust the `network_type` variable in `lib/config.py`. Also note that in `split_test_coins.py`, parameters like private key in Bitcoin or BlockCyper testnets in `config.py` are already imported. Check those imported parameters at the beginning of `split_test_coins.py` and select corresponding parameters to put them at the end of `split_test_coins.py`.

8. At the end, verify that you created Bitcoin Testnet addresses for you, Alice, and Bob in a block explorer (e.g. https://live.blockcypher.com/ and select Bitcoin testnet). You and Alice should have some coins on this blockchain. There should also be BlockCypher Testnet addresses for Alice and Bob. Bob should have some coins on this blockchain. You can check by using a block explorer (e.g. https://live.blockcypher.com/ and select BlockCypher Testnet). If you record all the

results properly in a separate file as suggested in the previous step, you should have no problem finding the corresponding address of a user in a specific testnet. Give yourself a pat on the back for finishing a long setup. Now it's time to explore creating transactions with Bitcoin Script.

4 Questions

For each of the questions below, you will use the Bitcoin Script opcodes to create transactions. For question 4, you will write an atomic swap transaction across two different blockchains. To publish each transaction created for the exercises, edit the parameters at the bottom of the file to specify which transaction output the solution should be run with along with the amount to send in the transaction. If the scripts you write aren't valid, an exception will be thrown before they're published. For questions 1-3, make sure to record the transaction hash of the created transaction and write it to `docs/transactions.py`. After completing each exercise, look up the transaction hash in a blockchain explorer to verify whether the transaction was picked up by the network, i.e., finalized with 6+ confirmations. Make sure that all your transactions have been posted successfully before submitting their hashes.

Exercise 1. Open `Q1.py` and complete the scripts labelled with `TODO`s to redeem an output you own and send it back to the faucet with a standard PayToPublicKeyHash transaction. The faucet address is already included in the starter code for you. Your functions should return a list consisting of only OP codes and parameters passed into the function.

Exercise 2. For question 2, we will generate a transaction that is dependent on some constants.

(a) Open `Q2a.py`. Generate a transaction that can be redeemed by the solution (x, y) to the following system of two linear equations:

$$x + y = \text{(YYYY, which is your birthday year)} \qquad \text{and}$$

$$x - y = \text{(MMDD, which is your birthday month and day without leading zero)}$$

For an integer solution to exist, the rightmost digit of your birthday year and your birthday month and day must either be both even or both odd. Therefore, you can change the rightmost digit of your birthday month and day to match the evenness or oddness of the rightmost digit of your birthday year. For example, if your birthday is January 1st, 2000. Then $x + y = 2000$ and $x - y = 101$. Now, to match even and oddness, change 101 to 102, so you have $x + y = 2000$ and $x - y = 102$. You then have an integer solution $x = 1051$ and $y = 949$. Make sure you use `OP_ADD` and `OP_SUB` in your scriptPubKey.

(b) Open `Q2b.py`. Redeem the transaction you generated above. The redemption script should be as small as possible. That is, a valid scriptSig should consist of simply pushing two integers x and y to the stack.

Exercise 3. Next, we will create a multi-sig transaction involving four parties.

(a) Open `Q3a.py`. Generate a multi-sig transaction involving four parties such that the transaction can be redeemed by the first party (bank) combined with any one of the 3 others (customers) but not by only the customers or only the bank. You may assume the role of the bank for this problem so that the bank's private key is your private key and the bank's public key is your public key. Generate the customers' keys using `lib/keygen.py` and paste them in `Q3a.py`.

(b) Open `Q3b.py`. Redeem the transaction and make sure that the scriptSig is as small as possible. You can use any legal combination of signatures to redeem the transaction but make sure that all combinations would have worked. Specifically, make sure that bank's signature plus any customer's signature should redeem the transaction. You do not need to worry about the case when multiple customers' signatures are provided. You can design the order of the signatures anyway you want as long as it can redeem the transaction.

Exercise 4. Last but not least, you will create a transaction called a *cross-chain atomic swap*, allowing two entities to securely trade ownership over cryptocurrencies on different blockchains. In this case, Alice and Bob will swap coins between the Bitcoin testnet and BlockCypher testnet. As you recall from setup, Alice has bitcoin on BTC Testnet3, and Bob has bitcoin on the BCY Testnet. They want to trade ownership of their respective coins securely, something that can't be done with a simple transaction because they are on different blockchains. The idea here is to set up transactions around a secret x, that only one party (Alice) knows. In these transactions only H(x) will be published, leaving x secret. Transactions will be set up in such a way that once x is revealed, both parties can redeem the coins sent by the other party. If x is never revealed, both parties will be able to retrieve their original coins safely, without help from the other. Before you start, make sure to read `swap.py`, `alice.py`, and `bob.py`. Compare to the pseudocode in https://en.bitcoin.it/wiki/Atomic_cross-chain_trading. This will be very helpful in understanding this assignment. Note that for this question, you will only be editing `Q4.py` and you can test your code by running `python swap.py`.

(a) Consider the ScriptPubKey necessary for creating a transaction to carry out a cross-chain atomic swap. This transaction must be redeemable by the recipient (if they have a secret x that corresponds to Hash(x)), or redeemable with signatures from both the sender and the recipient. Write this ScriptPubKey in coinExchangeScript in `Q4.py`.

(b) Write the accompanying ScriptSigs:

i. Write the ScriptSig necessary to redeem the transaction in the case where the recipient knows the secret x. Write this in coinExchangeScriptSig1 in `Q4.py`.

ii. Write the ScriptSig necessary to redeem the transaction in the case where both the sender and the recipient sign the transaction. Write this in coinExchangeScriptSig2 in `Q4.py`.

(c) Run your code using `python swap.py`. We aren't requiring that the transactions be broadcasted, as that requires some waiting to validate transactions. Running with broadcast_transactions=False will validate that ScriptSig + ScriptPK return true. Try this for alice_redeems=True as well as alice_redeems=False.

OPTIONAL: Try with broadcast_transactions=True, which will make the code sleep for an appropriate amount of time to post everything to the blockchain and verify correctly. Warning: will take 30 or more minutes to run.

(d) Fill in `docs/Q4design_doc.txt` with the following information:

i. An explanation of what you wrote and how the coinExchangeScript works.

ii. Briefly, how the coinExchangeScript you wrote fits into the bigger picture of this atomic swap.

iii. Consider the case of Alice sending coins to Bob with coinExchangeScript:

- Why can Alice always get her money back if Bob doesn't redeem it?
- Why can't this be solved with a simple 1-of-2 multisig?

5 Submitting your code

Record your transaction hashes in the `docs/transactions.py` file for questions 1-3. The hashes should be listed one per line in the same order as the questions.

For question 4, make sure `docs/Q4design_doc.txt` is filled out and your code verifies when run with `broadcast_transactions=False`.

Please submit all code for this assignment. Please create a single .tar or .zip file that includes all your deliverables for all four questions. Submit via Gradescope.

6 FAQ and tips

1. When you run `keygen.py` during setup, you may encounter the following libeay32 issue in the error message.

```
FileNotFoundError: Could not find module 'libeay32' (or one of its
dependencies). Try using the full path with constructor syntax.
```

This seems to happen in Windows OS very often. The reason is that the following command `_ssl = ctypes.cdll.LoadLibrary(ctypes.util.find_library('ssl') or 'libeay32')` in `key.py` can not find `libeay32`.

For Windows OS, one fix used by some students in the past is to directly download `libeay32` in `openssl-1.0.2d-x64_86-win64.zip` from https://indy.fulgan.com/SSL/Archive/. Then add `libeay32.dll` to System32 folder directly.

Another fix used by some students in the past is to install anaconda and use conda environment. First, install anaconda https://www.anaconda.com/products/distribution. Anaconda includes conda. Then create and activate conda environment using `conda create -n myenv python=3.9`, then `conda init cmd.exe`, and `conda activate myenv`. For details, see https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html. Finally, do `pip install -r requirements.txt`.

2. You can check recommended reading 1 for the available opcodes. You should not use the opcodes that are disabled because miners will reject the script with disabled opcodes.

3. In general, you should provide a reasonable transaction fee because the transaction fee may determine the priority. You can find current fee estimates on BlockCypher's block explorer below:

   BTC Testnet: https://live.blockcypher.com/btc-testnet/

   BCY Testnet: https://live.blockcypher.com/bcy/

4. At the end of the script, the top of the stack must be 1 or True. Hence, if you consume the last value, it won't work. See the following quote in recommended reading 1 `A transaction is valid if nothing in the combined script triggers failure and the top stack item is True (non-zero) when the script exits.`

5. It is a good practice to verify that your scripts are fully correct before you submit a transaction. This means that when the proper inputs are provided, the script should terminate with only one item on the stack, and that item should be True. The following link provides visualization of bitcoin script execution, which may be useful. https://siminchen.github.io/bitcoinIDE/build/editor.html

6. Negative numbers can go into the stack. The python encoding of an integer should be able to be pushed onto the stack with no extra work from you, even if it's negative.

7. For problem 3(b), you should have a shorter script if you use `OP_CHECKMULTISIG` somewhere in the `ScriptPK`.

8. f you mess up the UTXO order (could be 0 or 1 if you have two outputs), then you may have the following error message)

```
400 Bad Request
{"error": "Error validating transaction: witness script detected
in tx without witness data."}
```

9. In the past, there were questions regarding what is being asked for the writeup in 4(d). Further elaboration is provided below.

   i What does your code do? How does it implement the specification we set out for you?

   ii Look through the atomic swap protocol on the wiki. What part(s) of the protocol does your code represent? What does the code you wrote do in the protocol and why is it necessary?

   iii Why can Alice always get her money back if Bob does not comply with the protocol? You may discuss the entire protocol, not just the part you implemented yourself. Why can't Alice instead issue a 1-out-of-2 multisig with her and Bob's public keys and use this transaction to get a refund if Bob does not comply instead of the existing mechanism?

7 Recommended Reading

1. Bitcoin Script: https://en.bitcoin.it/wiki/Script
2. Bitcoin Transaction Format: https://en.bitcoin.it/wiki/Transaction
3. Bitcoin Transaction Details: https://privatekeys.org/2018/04/17/anatomy-of-a-bitcoin-transaction/
4. How Atomic Swap Works: https://en.bitcoin.it/wiki/Atomic_cross-chain_trading