Lab

Python Ethereum & Debugging Py-EVM

Create a new environment and build the environment in it. Such as a VM in

- **Leaning Objectives**
    - Introduction into the test “IDE”
    - Create a simple test program.
    - Send transaction with py-evm
    - PoW vs. Consensus
    - PoW Analysis and Understanding
    - Extensively go through the available commands in the py-evm environment.
    - Build a debugging virtual Environment to test Py-EVM
    - Virtual Environment
    - Run File vs Interactive Shell python
- **Equipment/Software**
    - Ubuntu 20.04 LTS Desktop
    - Internet connection
    - Python3
    - py-evm
    - pip3
    - git
    - Vim

- **Exercises**
    - **Exercise 1: Python Ethereum Introduction, Setup & Transaction**
    - **Exercise 2: PoW introduction, PoW example, PoW logs**
    - **Exercise 3: Introduction to Debugging Py-EVM**
    - **Exercise 4: Advanced Py-EVM debugging**
    - **Step Extra Credit: Real World Esque PoW**
- **Python Ethereum**
- **Exercises:**
    - **Exercise 1: Python Ethereum Introduction, Setup & Transaction**
        - **Step 1: Ethereum Platform: PY-EVM**
            - **T0: Ethereum Intro**
                - Ethereum is a platform that is decentralized in nature, runs as a peer to peer network, that is used as a public ledger of everything that happens on the platform.
                - On this platform we have accounts, full nodes, authenticators, uncles, etcand we interact with each other by sending a transaction to another account. A transaction is an exchange of value between two accounts. We chose Ethereum for a specific reason, as it has many different transaction types, and it is not based on a cryptocurrency architecture.

- Ethereum is way more capable than other contemporaries as we can exchange anything of value, such as logs, receipts, information etc…
- This lab is intended as a very high overview of how the platform operates, and how we can write code to get values out of the interactive shell

- **T1: Environment Setup**
    - To start we need to prepare our environment. To do this run the following commands:
        - ***sudo apt update***
        - ***sudo apt upgrade***
    - We will need a folder to house all our folders, create a folder called "*py-blkchain*" and navigate to the newly created directory.
    - We want to build this in a virtual environment for purposes of this lab, to do this execute the following commands
        - ***pip install virtualenv***
        - ***virtualenv -p python3 venv***
        - ***. venv/bin/activate***
    - Now we have prepared our machine we can start downloading dependencies
        - ***sudo apt-get install python3.9-dev***
        - ***sudo apt-get install python3-pip***

-

```
checkout@boot-node:~/python-blkchain$ . venv/bin/activate
(venv) checkout@boot-node:~/python-blkchain$
```

    - Now we need to make sure pip is up to date so we can install the py-evm, and we install the py-evm
        - ***pip3 install -U pip***

```
(venv) checkout@boot-node:~/python-blkchain$ pip3 install -U pip
Requirement already satisfied: pip in ./venv/lib/python3.8/site-packages (22.0.4
)
(venv) checkout@boot-node:~/python-blkchain$
```

        - ***pip3 install -U py-evm***
    - We now have all the things to run ethereum on our Ubuntu VM. Next we are going to explore how to interact with the python interactive console.

- **Q1: What do the above commands do?**
- **A1: The above commands to the following**
    - ***sudo apt-get update && sudo apt-get upgrade***
        - **This installs the newest packages for all existing packages installed on the machine and upgrades them to the newest version**
    - ***sudo apt-get install python3.9-dev && sudo apt-get install python3-pip***
        - **These commands install the python 3.9-developers environment, and the second**

command installs pip for python3. Pip is like apt, and an install packages and other things.

- ***pip install virtualenv && virtualenv -p python3 venv && . venv/bin/activate***
  - **In the first command we install the "virtual environment" package for python.**
  - **The second command we create a virtual environment called "venv".**
  - **The third command activates the virtual environment. Confirmation of this can be seen in the CLI, it now has (venv) to the left of the "checkout@ubuntu" string. This indicates we are in the virtual environment.**
- ***pip3 install -U pip && pip3 install -U py-evm***
  - **The first command installs the pip package, this can be used to install other packages. Like Vim for Vi, pip3 is to pip, where we are getting an older version of the application.**
  - **In the second command we are installing the python ethereum virtual machine, the vm is the "py-evm" package.**

- **T2: Python Virtual Environment**

- Python can be coded in a virtual environment, this is why we activated the “venv” virtual environment. We will operate in the virtual environment for the remainder of the lab. This is because of the packages that are contained in the virtual environment called, “venv”.

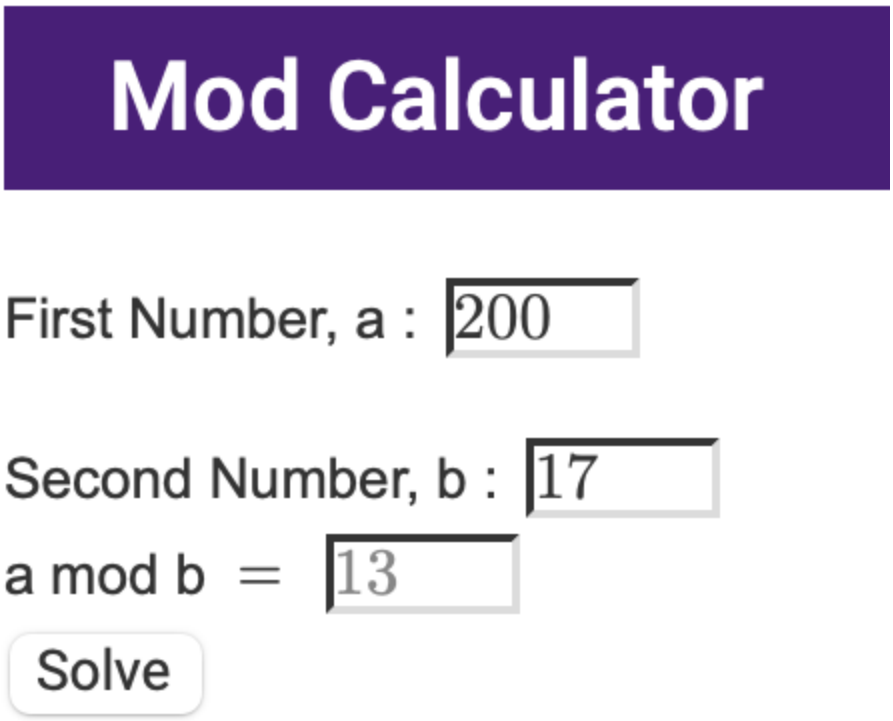
- **Q2: List the differences between an interactive shell and a normal shell. Give an example for using each shell over the other.**
- A2: The differences between the two environments is that a virtual environment is an environment where we can store dependencies that are not tied to the normal python environment you run. This is done so you can keep 2 or more different environments depending on your needs for the language you are using. In this lab, we created the virtual environment, and installed the py-evm, python3.9-dev and the rest of the packages. From the commands we can glean that we will be using python 3.9 for the py-evm, while we are still running python 3.8 on the host machine. This will allow us to keep all python 3.8 and 3.9 applications separate.

- **T3: Running a simple python application in virtual environment**

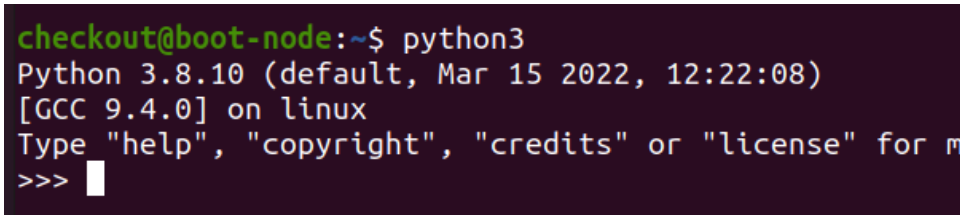
- With the virtual environment, we will run an application from the command line. Open up vim and copy Appendix A into the file, name the file E1S1T3Q3.py and save it into new directory called "sample_files". (include this in your lab report). Make sure when you are copying Appendix A into vim, to use 4 spaces instead of a tab character.
- **Q3: What is the output of the program? Is this what you expected?**
- **A3: the output of the program is 13. This was what we were expecting, we double checked our math with an online modulo calculator.**

```
q = 20
w = 10
s = 17
tot = (q * w) % s
print(tot)
~
~
```

```
checkout@boot-node:~/python-blkchain/sample_files$ python3 E
13
checkout@boot-node:~/python-blkchain/sample_files$
```



- **T4: Python File from the Interactive Environment**
  - Now that we have run a python file from the command line we are going to explore the interactive shell.
  - To launch an interactive shell, type
    - **python3**
  - This will launch an interactive shell. We can confirm this by the changing of the command prompt. It should reflect ">>>" instead of the "checkout@ubuntu" that is in the normal CLI.



- Now to run a program from this interactive shell we enter in the variables and functions line by line. Enter in Appendix A line by line to run the program in interactive mode

```
checkout@boot-node:~$ python3
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more
>>> q = 17
>>> w = 10
>>> s = 10
>>> w = 20
>>> tot = (w * s) % q
>>> print(tot)
13
>>>
```

- **Q4: Is the output the same as before? Is this expected?**
- **A4: The output is 13, the same as before. This is expected because it is the same python code, but a different way to run it.**
- **Q5: What is the difference between the running a python program from the cli and running a python program in interactive mode?**
- **A5: The difference between calling a file from the command line to run and the interactive shell is the fact that in the file call, the file runs and exits, where as the interactive mode the file is always running, and we can add more variables and functions line by line.**

Interactive mode is similar in function to the Jupyter Notebooks in terms of how they function.

- **S2: Simple App**
  - **T1**
    - In this task we will download an example application from the ethereum-python project, this is a simple app used to show how we can use the py-evm and its interactive console to create things right on the blockchain.
    - We need to make a directory to hold all of our python ethereum code. This is so we can keep the Geth and python environments separate, even though they do the same thing (interact on the Ethereum platform), but the python console is easier to write code in and get information you cannot get from Geth methods.
    - The python environment is good for the undergraduate level as they will probably have more experience with python than they do with javascript, which Geth runs on.
    - Make your python ethereum file, navigate to it, create another file to hold our app with the following command
      - ***cd python-blkchain***
    - Since we are using an example project, we are going to clone it from github with the following command

- ***git clone https://github.com/ethereum/ethereum-python-project-template.git first-app***

```
checkout@boot-node:~/python-blkchain$ git clone https://github.com/ethereum/ethereum-python-project-template.git demo-app
```

- Navigate to the folder that was created

```
checkout@boot-node:~/python-blkchain$ cd demo-app/
```

```
checkout@boot-node:~/python-blkchain/demo-app$ ls
docs        MANIFEST.in    pyproject.toml  requirements-docs.txt  tox.ini
LICENSE     mypy.ini       pytest.ini      setup.py
Makefile    newsfragments  README.md       tests
checkout@boot-node:~/python-blkchain/demo-app$
```

```
checkout@boot-node:~/python-blkchain/demo-app$ sudo vim setup.py
```

- **T2**
  - We now need to edit the setup.py file of the app and change the install requirements to fit our app and environment compatibility
    - Change the following in the *setup.py* script
      - In the “install_requires” section edit it to say the following:
        - “eth-utils>=1,<2”,
        - “py-evm==0.5.0a0”,

```
    install_requires=[
        "eth-utils>=1,<2",
        "py-evm==0.5.0a0",
    ],
```

- In the “name” we need to change that too or else we will get errors in the next tasks, change the name to “**CapstoneFirstApp**”

```
setup(
    name='CapstoneFirstApp',
```

- **T3**
    - Next we need to install dependencies
        - ***pip install -e “.[dev]”***

```
  Running setup.py develop for CapstoneFirstApp
Successfully installed CapstoneFirstApp-0.1.0a0
```

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ ls
CapstoneFirstApp.egg-info  LICENSE   MANIFEST.in  newsfragments   pytest.ini  requirements-docs.txt  tests
docs                       Makefile  mypy.ini     pyproject.toml  README.md   setup.py               tox.ini
(venv) checkout@boot-node:~/python-blkchain/demo-app$ mkdir app
```

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ cd app
(venv) checkout@boot-node:~/python-blkchain/demo-app/app$
```

Open main.py ~/python-blkchain/demo-app/app

```
>>> from eth import constants
>>> from eth.chains.mainnet import MainnetChain
>>> from eth.db.atomic import AtomicDB

>>> from eth_utils import to_wei, encode_hex


>>> MOCK_ADDRESS = constants.ZERO_ADDRESS
>>> DEFAULT_INITIAL_BALANCE = to_wei(10000, 'ether')

>>> GENESIS_PARAMS = {
...     'difficulty': constants.GENESIS_DIFFICULTY,
... }

>>> GENESIS_STATE = {
...     MOCK_ADDRESS: {
...         "balance": DEFAULT_INITIAL_BALANCE,
...         "nonce": 0,
...         "code": b'',
...         "storage": {}
...     }
... }

>>> chain = MainnetChain.from_genesis(AtomicDB(), GENESIS_PARAMS, GENESIS_STATE)

>>> mock_address_balance = chain.get_vm().state.get_balance(MOCK_ADDRESS)

>>> print("The balance of address {} is {} wei".format(
...     encode_hex(MOCK_ADDRESS),
...     mock_address_balance)
... )
The balance of address 0x0000000000000000000000000000000000000000 is 10000000000000000000 wei
```

    - Now we need to enter the python interactive shell to show how we can write code and interact with the Ethereum VM we just installed and cloned an app for.

- **T4: Write App Code & Run**
    - We are going to write an application that is going to show how we can send ether from one account to another, like is Alice wanted to send Bob some ether. We will build on this code as we continue.
    - Repeat line by line the commands listed in Appendix A in the python interactive shell.

- **Q1: Once complete what is the output of the last command?**
- **A1: the output was: "The balance of address 0x00000000000000000000000000 is 10000000000000000"**

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ python app/main.py
The balance of address 0x0000000000000000000000000000000000000000 is 10000000000000000000 wei
(venv) checkout@boot-node:~/python-blkchain/demo-app$
```

- Now that we have seen how we can interact with the interactive console, we can also run the same program from the "venv" cli. To do this run the following command
  - ***python app/main.py***
- **Q2: What is the output of this command? Is it the same as the output from the interactive shell? Is this what we expected?**
- **A2: the output was: "The balance of address 0x00000000000000000000000000 is 10000000000000000"**
- **Q3: Why is this helpful, if we can enter in commands two different ways? Is this helpful for things other than running apps?**
- **A3: This is helpful because we can create testing applications and run them on the system at any time we want. This allows us to check values in the py-evm with**

simple python scripts, which we normally wouldn't be allowed to see their contents.

- **Step 3: Blockchain Functionality**
  - Now that we have seen that we can send a transaction with this platform between two user, lets build an application that will handle the chain, block and validation part of the blockchain ether value exchange.
  - In this instance we are going to have Alice send Bob a transaction like we did before, but now we are going to create a new chain (this is done for repeatable values every time we run the code), sign a transaction, have the transaction validated, and sealed in a block that gets added to the blockchain.
  - **T1: Send Transaction**
    - Unlike Geth and Metamask, where you need to go in, setup accounts, and make sure your environment is configured properly, the py-evm is meant to conduct tests on the platform. This means it is really lightweight if you want it to be, which is good for exploring the inner workings of the platform
    - We are going to take advantage of this ability of the py-evm, in conjunction with the python interactive console

- We are going to follow the commands in Appendix B to setup a test chain, a sender private key, receiver, compose and sign a transaction.
- Once done with running the above in the following interactive console, we are able to validate (mine) the block

```
(venv) checkout@boot-node:~$ python
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from eth import constants, Chain
>>> from eth.vm.forks.frontier import FrontierVM
>>> from eth.vm.forks.homestead import HomesteadVM
>>> from eth.chains.mainnet import HOMESTEAD_MAINNET_BLOCK
>>> chain_class = Chain.configure(
...     __name__='Test Chain',
...     vm_configuration=(
...         (constants.GENESIS_BLOCK_NUMBER, FrontierVM),
...         (HOMESTEAD_MAINNET_BLOCK, HomesteadVM),
...     ),
... )
>>> from eth.db.atomic import AtomicDB
>>> from eth.chains.mainnet import MAINNET_GENESIS_HEADER
>>> #start a fresh in-memory db
>>> #initialize fresh chain
>>> chain = chain_class.from_genesis_header(AtomicDB(), MAINNET_GENESIS_HEADER)
>>>
```

- **T2: Block Mining**
  - Follow the commands in Appendix C to mine the block and add it to the chain
  - Once done, we are ready to get into how we validated (mined) this block.

```
>>> chain = klass.from_genesis(AtomicDB(), GENESIS_PARAMS)
>>> from eth.consensus.pow import mine_pow_nonce
>>> block_result = chain.get_vm().finalize_block(chain.get_block())
>>> block = block_result.block
>>> nonce, mix_hash = mine_pow_nonce(
...     block.number,
...     block.header.mining_hash,
...     block.header.difficulty)
>>> block = chain.mine_block(mix_hash=mix_hash, nonce=nonce)
>>> print(block)
Block #1-0xcd77..c733
```

- **Step 2: PoW introduction, PoW example, PoW logs (Appendix mbe)**
  - **T0: Consensus vs. PoW?**
    - So we have now sent a transaction and mined the block that contained the transaction. But what is mining/block validation?
    - Ethereum 1.0 runs with a PoW consensus algorithm. This is a brute force system to find a nonce value with a mix_hash to create a hash that is less than the threshold hash. The threshold hash value is taken from the block header values.
    - Essentially, what validation broken down is a lot of computational power going into trying to find a value that is lower than a predetermined value.

- **Step 1: Simple Python PoW example**
  - **T1: Code Analysis & Understanding**
    - Download the following file:
      - *simple_consensus.py*

- Open the file and take a look at the code
- **Q1: What is the hash we are looking for? What is the range we are looking for a number to solve the threshold hash?**
- **A1: The hash we are looking for is: 1000005. We are using the random function in python to search for a number between 1000000 and 1000000000.**
- **Q2: Is there a way we can look at the previous guesses the program has made? If so where? If not, how can we make a change to the code to print out the guesses?**
- **A2: In the code there is a line that opens a file and writes to it, but it is currently commented out. If we want to see the guesses, we can uncomment the line and read from the text file after running the code.**

- **T2: PoW implementation**
  - Now that we have taken a look at the code we are going to run it. We will be greeted with statistics from the algorithm's performance.

- **Q3: How long did it take for your program to execute? Is the execution time the same as the time it took the process to run the program, or is it defined in the code around specific parameters?**
- **A3: It took around 3 seconds for the program to solve the hash. In the code, we start the timer before each function and stop it at the end of each function, so it really is timing each process individually.**
- **Q4: How long did it take for the computer take to "check" the guess?**
- **A4: it took fractions of a second.**

- **T3: PoW Across Network**
  - When a miner validates a block a broadcast message is sent out to its neighbor nodes to check its work. Ethereum is check, then verify type of system.

- **Q5: Is the delta between the find hash time and the check time significant? If so, why? If not, why not?**
- **A5: The delta, when viewed by a human is not significant, but when compared to each other, the checking is quicker by a significant margin. This is because when we check we are just validating the values that have been reported to have solved the hash.**
- **Q6: How does these results explain why blocks are extremely rarely left behind?**
- **A6: The reason for this is the extremely quick, compared to the find hash part, checking function. Since we already know the value reported by the miner that "won" the block with. As a peer on the network, we do not need to find the "winning" hash, we just need to check the values that reportedly solved the hash, work on our machine, once that happens the node checking is in consensus with the other node, if correct. The quick propagation time aides in the spread of this information across the network as the other peers are trying to solve the current block.**
- **Q7: What does nonce represent in this simple code? How many guesses did it take to find the guess that solved the puzzle?**

- **A7: the nonce is the number of guesses we have tried.**

  The number of guesses is 2286252.

- **Ex3 Step 1: Py-EVM Intro**

  - **T2: Py-EVM Commands**

    - Now that we have seen the python code work on the VM, in the interactive shell, and with the python3 command, we should explore more of the functions we can interact with in the interactive console of the py-evm.
    - As a developer exploring Ethereum's platform, the more we can understand directly from the command line, the better we are going to understand the inner workings of the py-evm.
    - Enter the following to get the block number
      - **chain.get_block()**

```
>>> chain.get_block()
```

      - **state.get_balance(SOME_ADDRESS)**

```
>>> state.get_balance(SOME_ADDRESS)
1000000000000000000000
```

      - **ByzantiumVM.get_block_reward()**

```
>>> ByzantiumVM.get_block_reward()
3000000000000000000
```

- **vm.previous_hashes**

```
>>> vm.previous_hashes
<generator object VM.get_prev_hashes at 0x7f0e295050b0>
```

- **Q8: What is the output of the previous_hashes function? What does the output mean? Is there any way we can see the contents of it?**
- **A8: the output of the command is "<generator object VM.get_prev_hashes at 0x7f0e295050b0>". This is telling us where the generator is located in the memory, but we cannot print out the values. If there was a way we could return the values inside a variable, we should be able to print out the variable to see the values.**
- **Q9: We have been over the py-evm decently well, what can we do to see the values of the previous guesses of the block?**
- **A9: Since we are working with python, I know when in normal python I get this error I can cast to a tuple and see the values, so I created a variable 'j', then I print j from inside the interactive python terminal. Since it is a tuple, we can use all the methods tuples have access to**

in vanilla python. Such as the length of the tuple j, which in my case was 2.

```
>>> j = tuple(vm.previous_hashes)
>>> print(j)
(b'\xcdw@\x10\xc9E2\x94\x81~\xdb\xd4\xbfZoI\xe8\xa6\x93\xda\xbc\xdd\xb1+\x96\x0ex\xef|\x1d\xc73', b'\xac \xab\x10\xe9\x8f\xe0\xf6UL\xb3$\x97\xbc\xe8c\xbe\x
03-_\x8c\xb0{\x15\tq3_\xb2\x93\x8a\xf1')
>>> k = chain.get_block()
>>> print(k)
Block #3-0x4b14..0c85
>>> print(len(j))
2
```

- **Exercise 3: Introduction to Debugging Py-EVM**
  - **T0: Describe a scenario, we then will go down to the code to explain the code. We need to have the introduction to the scenario, we need to have a consumer view (Alice wants to send Bob a transaction, these things are)**
    - **Focus more on the consumer**
  - **T1: Blockchain Generation**
    - In this task we are going to need to copy Appendix E into the file where we created the demo application in the exercise before.

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ python3 app/mai
**********************************************************************
*************************** Welcome to the py-evm logger *************
**********************************************************************
(venv) checkout@boot-node:~/python-blkchain/demo-app$
```

    - Once you have copied the contents of Appendix E, we are going to need to uncomment some lines of code. Use Appendix F as a guide for which lines to comment out for each different task.

```
38 print("*********************************************************************************")
39 print("************************** Welcome to the py-evm logger ***************************")
40 print("*********************************************************************************")
41 print("Part 1")
42 print("*********************************************************************************")
43 print("We just created the chain, here it is: ", chain)
44 print("We just created the mining chain object klass, here it is: ", klass)
45 print("We just created the genesis block, here it is", genesis)
46 print("We just created the VM, here it is: ", vm)
47
```

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ python3 app/main.py
*********************************************************************************
************************** Welcome to the py-evm logger ***************************
*********************************************************************************
Part 1
*********************************************************************************
We just created the chain, here it is:  <abc.TestChain object at 0x7f1cab23f9a0>
We just created the mining chain object klass, here it is:  <class 'abc.TestChain'>
We just created the genesis block, here it is <BlockHeader #0 26a31198>
We just created the VM, here it is:  <eth.vm.forks.byzantium.ByzantiumVM object at 0x7f1cab2a8d60>
(venv) checkout@boot-node:~/python-blkchain/demo-app$
```

Q1: Describe each line of the output for the block creation part of the code.

A1: In the first line we have a pointer to a location in memory where the chain object is located. In the second line we created a mining chain for block validation, this line tells us what class the variable is. In the third log we can see that we are on the first block, the genesis, #0. The numbers in the block header are from the environmental variables that are used to create a hash for the upcoming block.

- **T2: Transaction Generation**
  - Uncomment the lines according to appendix F.
  -

```
66
67 print("*********************************************************************************")
68 print("*********************************************************************************")
69 print("*********************************************************************************")
70 print("Part 2")
71 print("*********************************************************************************")
72 print("This is the the SENDER variable", SENDER)
73 print("This is the result of the Address object for the receiver: ", RECEIVER)
74 print("This is the sender's private key, this is it decoded", SENDER_PRIVATE_KEY)
75 print("this is the unsigned transaction", tx)
76 print("This is the user's nonce:  ",nonce)
77 print("this is the signed transaction", signed_tx)
78
```

- ```
  (venv) checkout@boot-node:~/python-blkchain/demo-app$ python3 app/main.py
  ******************************************************************************
  ************************* Welcome to the py-evm logger ************************
  ******************************************************************************
  ******************************************************************************
  ******************************************************************************
  ******************************************************************************
  Part 2
  ******************************************************************************
  This is the the SENDER variable b'\xa9OSt\xfc\xe5\xed\xbc\x8e*\x86\x97\xc1S1g~n\xbf\x0b'
  This is the result of the Address object for the receiver:  b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
  This is the sender's private key, this is it decoded 0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
  this is the unsigned transaction ByzantiumUnsignedTransaction(nonce=0, gas_price=0, gas=100000, to=b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
  x00\x00\x00\x02', value=0, data=b'')
  This is the user's nonce:   0
  this is the signed transaction ByzantiumTransaction(nonce=0, gas_price=0, gas=100000, to=b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
  0\x02', value=0, data=b'', v=28, r=31462166518758896569909184273588344974771178431451132490293541023460279591788, s=2241845836193961543958327
  934928793750824068103012324)
  (venv) checkout@boot-node:~/python-blkchain/demo-app$
  ```

- ```python
  SENDER_PRIVATE_KEY = keys.PrivateKey(
      decode_hex('0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065f
  )
  SENDER = Address(SENDER_PRIVATE_KEY.public_key.to_canonical_address())
  ```

  Q2: What is the sender's address? What created this address?

- **A2: The sender's address is:**

  b'\xa9OSt\xfc\xe5\xed\xbc\x8e*\x86\x97\xc1S1g~n\xbf\x0b', when we look at the code the we use the built in Address function to use the private key we created to create an address by mapping it to a canonical address. The private key is what is used to verify you transaction as it is the way to verify your account.

- **Q3: How many times did we send a transaction in this code? How many fields are there in the type of raw transaction? How many fields are are in the signed transaction?**

- **A3: we sent 1 transaction, but we did 2 things to create the transaction. We first had to create a transaction object, but this is a raw transaction. There is no confirmation that the sender and receiver exist and the amount of value to transfer has not been validated yet. When we apply tx (the raw transaction) with the signed transaction method, we see that some more data fields are populated**

in the transaction header. In the raw transaction fields we have a total of 6 fields (nonce, gas_price, gas, to, value, data). In the signed transaction we have a total of 9 fields (the 6 from before, but we now have fields, R, V, S.

- **Q4: What is the value of the nonce in this case(this is different from the mining nonce)? What will this affect if not handled properly in your code?**
- **A4: The value of the nonce is 0. This nonce is different than the nonce used in the PoW. This nonce keeps track of the amount of transactions the sender sends. When the sender has the same nonce as another unverified transaction (ie: two transactions in one block) then the second transaction will be "queued" instead of "pending".**
- Note, I am still looking for the python API equivalent of the txpool.status() in Geth.
- **T3: Block Creation, Packing**
    - Uncomment the lines according to appendix F.

```
print("*********************************************************************************")
print("*********************************************************************************")
print("*********************************************************************************")
print("Part 3")
print("*********************************************************************************")
print("This is the block type result we got: ", q)
print("This is the block type we got", w)
print("This is the nonce value from the mine_pow_nonce method: ", nonce)
print("This is the mix_hash value from the mine_pow_nonce method: ", mix_hash)
```

    -

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ python3 app/main.py
********************************************************************************
************************** Welcome to the py-evm logger ***************************
********************************************************************************
********************************************************************************
********************************************************************************
********************************************************************************
Part 3
********************************************************************************
This is the block type result we got:  <class 'eth.abc.BlockAndMetaWitness'>
This is the block type we got <class 'eth.vm.forks.byzantium.blocks.ByzantiumBlock'>
This is the nonce value from the mine_pow_nonce method:  b'\x00\x00\x00\x00\x00\x00\x00\xa0'
This is the mix_hash value from the mine_pow_nonce method:  b'ej\xa1\xbd\xd0\x9d\xce\xfe\x9ah\xe4(g\\Hu\xdd=\xd5<h\xc0\x9cDA\
(venv) checkout@boot-node:~/python-blkchain/demo-app$
```

- Note in this example we are not getting anything of that much importance here, but this is important to see before we go into the source code and start alerting the code of the Py-EVM. When we do that, we can see the actual block packing and creation. This is something that is nice to have before we do that to show how much it really does.

- **T4: Validation (PoW)**
  - Uncomment the lines according to appendix F.

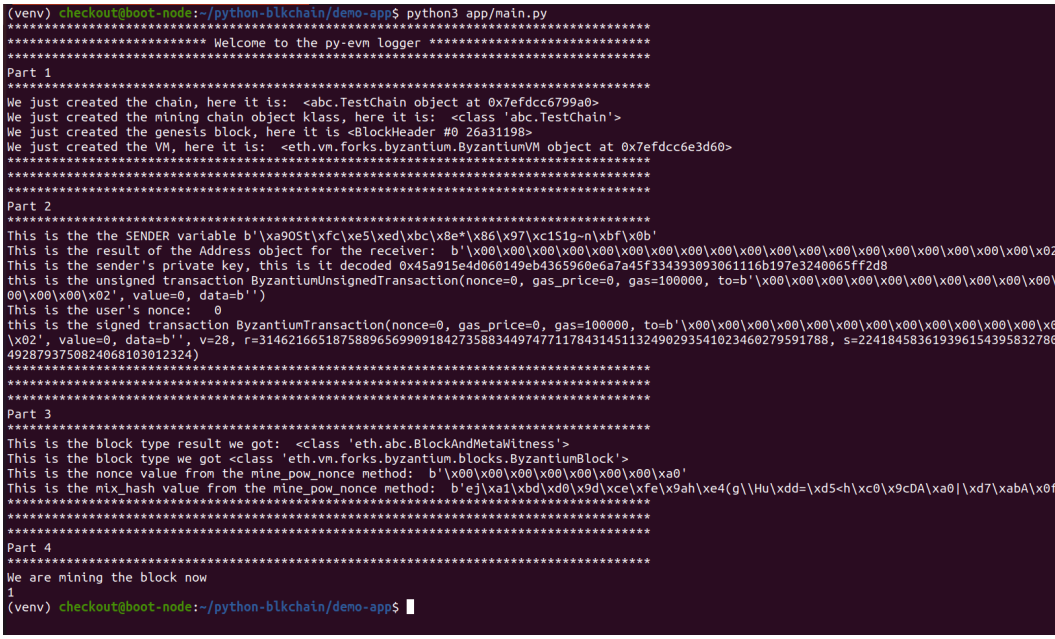
```
119
120 print("********************************************************************************")
121 print("********************************************************************************")
122 print("********************************************************************************")
123 print("Part 4")
124 print("********************************************************************************")
125 print("We are mining the block now")
126 print(block.number)
127
```

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ python3 app/main.py
********************************************************************************
************************** Welcome to the py-evm logger ***************************
********************************************************************************
********************************************************************************
********************************************************************************
********************************************************************************
Part 4
********************************************************************************
We are mining the block now
1
(venv) checkout@boot-node:~/python-blkchain/demo-app$
```

  - Note in this example we are not getting anything of that much importance here, but this is important to see before we go into the

source code and start alerting the code of the Py-EVM. When we do that, we can see the actual block packing and creation. This is something that is nice to have before we do that to show how much it really does.

- **T5: All uncommented**
    - Remove all comments from the main.py python file. This is the base of what our logs look like before we start to add logs to the source code.
    - Now that we know what the variables in the code are and how to get them with simple print statements in the code to show the value. In the next exercise we are going to be going into the source code of the Py-EVM to grab values in real time.



- **<u>Exercise 4: Advanced Py-EVM debugging</u>**

- In the previous exercises we have just played around with python, made a simple application, that runs a blockchain and we send a transaction between 2 users, validate the transaction, validate the block, then add it to the chain.
- In the previous exercises we went through the code to understand how the blockchain, transaction, account, block and mining are setup in the python code, and we know their values in case we want to trace them in this more advanced debugging stage
- Just like in the previous exercise, we are going to be using Appendix G
- **Step 1: Configure Environment**
    - If you have not done the previous exercises to setup the environment do them, this will allow you to understand how to read the logs, and what has been added to them that we do not get to see behind the scenes.
- **Step 2: Advanced Debugging, Py-Evm**
    - Now that we have done a cursory overview into how to write some debugging logs in a python file, it is time to do the same thing, but this python file is part of the source code
    - **T1: Validation.py debugging**
        - So we know that in the process of making an account, sending a transaction, signing a

transaction, Proof of Work and other things that occur behind the scenes need to be checked and validated before they are trusted. So we are going to see how many times we, “as the VM” we are going to validate something in the code.

- Refer to appendix G to know where to place the logs.
- Before running the app again, answer the following question.
- **Q1: What do you think these values are going to be? Do you think they will appear in other parts of the code because it is just checking values duing validation?**
- **A1: The values seem to be a comparison between maximum and value. If the value is lower than the maximum then the “validate_is_integer” is called, if it is higher than the maximum then a validation error is raised. This seems to be the part where the EVM checks numbers to see if the calculations are correct.**

```
95 def validate_lte(value: int, maximum: int, title: str =
96     #print("these are the values we are comparing in th
97     print("this is the guess: ", value)
98     print("this is the target: ", maximum)
99     if value > maximum:
100         raise ValidationError(
101             f"{title} {value} is not less than or equal
102         )
103     validate_is_integer(value, title=title)
104
```

- Run the main python app file again, like in the previous steps and exercises.
- **Q2: What are the results?**
- **A2: We get some new logs in both part 2 and part 4. Looking at the previous tasks in the exercise previous we know that Part 2 refers to the transaction creation and the signing of the transaction.**
- **Q3: Is there anything that appears in multiple places in Part 2?**
- **A3: Yes there is, we have repeat numbers in the signed transaction and the guess and target value logs. The numbers that repeat are and what they seem to be tied to (variable wise)**
    - **3146216651875889656990918427358834497477117843145113249029354102346027959178 → this is R, which we can see in the signed block transaction header.**

- 22418458361939615439583278042049262102873207836948934928793750824068103012324 → this is S, which we can see in the signed block transaction header
- 28 → this is V, which can be seen in the signed block transaction header
- 11579208923731619542357098500868790785326998466564056403945758400791312963399 → this one appears in both part 2 and part 3, which we know deals with the account and transaction validation, and Part 3, which deals with the PoW and block addition to the chain

- Q4: What is the value of the transaction? Is it located in the output? How can we tell that this is the transaction validation?
- A4: We can see that there is a transaction that has been verified when we compare the numbers:
- 22418458361939615439583278042049262102873207836948934928793750824068103012324 and: 57896044618658097711785492504343953926418782139537452191302581570759080747167.

-

```
********************************************************************************
********************************************************************************
********************************************************************************
Part 3
********************************************************************************
This is the block type result we got:  <class 'eth.abc.BlockAndMetaWitness'>
This is the block type we got <class 'eth.vm.forks.byzantium.blocks.ByzantiumBlock'>
This is the nonce value from the mine_pow_nonce method:  b'\x00\x00\x00\x00\x00\x00\x00\xa0'
This is the mix_hash value from the mine_pow_nonce method:  b'ej\xa1\xbd\xd0\x9d\xce\xfe\x9ah\xe4(g\\Hu\
this is the guess:  375262668542277611369857454100717638956796751024625434886276097325634339919
this is the target:  115792089237316195423570985008687907853269984665640564039457584007913129639
********************************************************************************
********************************************************************************
```

-

```
Part 2
********************************************************************************
This is the the SENDER variable b'\xa9OSt\xfc\xe5\xed\xbc\x8e*\x86\x97\xc1S1g~n\xbf\x0b'
This is the result of the Address object for the receiver:  b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
This is the sender's private key, this is it decoded 0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff
this is the unsigned transaction ByzantiumUnsignedTransaction(nonce=0, gas_price=0, gas=100000, to=b'\x00\x00\x00\x0
00\x00\x00\x02', value=0, data=b'')
This is the user's nonce:   0
this is the signed transaction ByzantiumTransaction(nonce=0, gas_price=0, gas=100000, to=b'\x00\x00\x00\x00\x00\x00
\x02', value=0, data=b'', v=28, r=31462166518758896569909184273588344974771178431451132490293541023460279591788, s=2
49287937508240681030123 24)
this is the guess:  31462166518758896569909184273588344974771178431451132490293541023460279591788
this is the target:  115792089237316195423570985008687907852837564279074904382605163141518161494336
this is the guess:  22418458361939615439583278042049262102873207836948934928793750824068103012324
this is the target:  115792089237316195423570985008687907852837564279074904382605163141518161494336
this is the guess:  28
this is the target:  28
this is the guess:  22418458361939615439583278042049262102873207836948934928793750824068103012324
this is the target:  57896044618658097711785492504343953926418782139537452191302581570759080747167
```

■ T2: Transaction Creation

- Refer to appendix G to know where to place the logs.

```
(venv) checkout@boot-node:~/python-blkchain/demo-app$ python3 app/main.py
********************************************************************************
************************* Welcome to the py-evm logger *************************
********************************************************************************
Part 1
********************************************************************************
We just created the chain, here it is:  <abc.TestChain object at 0x7f16870c6970>
We just created the mining chain object klass, here it is:  <class 'abc.TestChain'>
We just created the genesis block, here it is <BlockHeader #0 26a31198>
We just created the VM, here it is:  <eth.vm.forks.byzantium.ByzantiumVM object at 0x7f168712edc0>
These are the transaction parts:  [b'', b'', b'\x01\x86\xa0', b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
Our VRS output:  (28, 31462166518758896569909184273588344974771178431451132490293541023460279591788, 22418458361939
68103012324)
********************************************************************************
********************************************************************************
********************************************************************************
```

- **Q5: Where are the logs produced from the transaction.py file? What do the logs say?**
- **A5: The logs are in Part 1. We can see that there is a transaction with parts that are empty, which is similar to when we created the raw transaction. So this is before the transaction has been verified. We also See VRS output, this is**

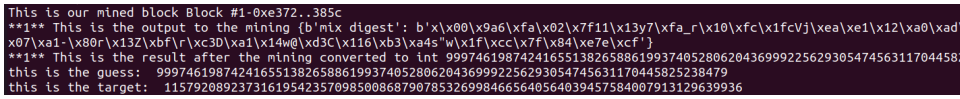
when the user gets validated, they will produce V, R, S values during the validation (these are unique for the user and not for the transaction).

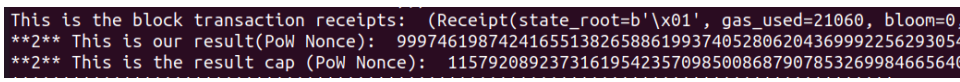
```
57
58 def create_transaction_signature(unsigned_txn: Unsign
59                                   private_key: datatyp
60                                   chain_id: int = None
61
62     transaction_parts = rlp.decode(rlp.encode(unsigne
63     print("These are the transaction parts:  ", trans
64     if chain_id:
65         transaction_parts_for_signature = (
66             transaction_parts + [int_to_big_endian(ch
67         )
68     else:
69         transaction_parts_for_signature = transaction
70
71     message = rlp.encode(transaction_parts_for_signat
72     signature = private_key.sign_msg(message)
73
74     canonical_v, r, s = signature.vrs
75
76     if chain_id:
77         v = canonical_v + chain_id * 2 + EIP155_CHAIN
78     else:
79         v = canonical_v + V_OFFSET
80     print("Our VRS output: ",VRS((v, r, s)))
81     return VRS((v, r, s))
82
```

- **T3: base.py**
  - Refer to Appendix G on how to comment the lines in this task
  - After you add your own logs to the source code of the Py-EVM run the main file, if you have done the editing correctly the program will run like normal. If not the Py-EVM is very easy to debug, we will take a look at this in the Extra Credit Section

- **Q6: What is the result? In what part are the logs you created located? What did the logs print out?**
- **A6: We find there is output in Part 3. We see that we have a new block header, and we print out the new block header. Then we grab the current block (the one we stopped accepting transactions for) and print it out in tuple form for easier readability. In the current block header we can see the numbers that appear in the guess and target comparison logs which appear in Part 2. We can also see that there is a signed transaction in the block (when we look at the fields in the block) which we are expecting because we created a raw transaction, signed it and it was included in this block.**

- **T4: PoW**

- Refer to Appendix G on how to comment the lines in this task
- After you add your own logs to the source code of the Py-EVM run the main file, if you have done the editing correctly the program will run like normal. If not the Py-EVM is very easy to debug, we will take a look at this in the Extra Credit Section
- **Q7: What is the result? In what part are the logs you created located? What did the logs print out?**
- **A7: From the python code we are able to see the difference between the PoW and the Validation of records. This is done in my logs with the "**1**" and "**2**" flags I put in the log output statement. These were both were put in the same file (pow.py) so they are being called from different points, but since this is part of the consensus algorithm Ethereum Uses and shows how everything is validated in everything we do**
- 

- 

- **Step Extra Credit: Real World Esque PoW**
  - T1: Leave the loggers in the code you have created. We are now going to start to change variables in the code for

the app and see them change in the code when we are running all of the loggers.

- Go to the main.py file that is the main file for our app we have been using to test the Py-EVM and its functionalities and logging certain variables to see the inner workings.
- We are going to edit the difficulty of the blockchain, we currently have it at 1. This is not realistic and provides no challenge for the PoW and provides no real world extensive logging data from the logging system we have built.
- **Q8: Go to the main python file and change the genesis difficulty value from 1 to the following values: 10, 100, 999, 1001. Describe the process for each of the 4 different difficulties. Show evidence in our logger that shows we have changed something in the blockchain configuration.**
- **10**

```
10
11
12 GENESIS_PARAMS = {
13     'difficulty': 10,
14     'gas_limit': 3141592,
15     # We set the timestamp, just to make this documented example rep
16     # In common usage, we remove the field to let py-evm choose a re
17     'timestamp': 1514764800,
18 }
19
```

- **100**

```
11
12 GENESIS_PARAMS = {
13     'difficulty': 100,
14     'gas_limit': 3141592,
15     # We set the timestamp, just to make this documented example r
16     # In common usage, we remove the field to let py-evm choose a
17     'timestamp': 1514764800,
18 }
```

- **999**

```
10
11
12 GENESIS_PARAMS = {
13     'difficulty': 999,
14     'gas_limit': 3141592,
15     # We set the timestamp, just to make this documented example r
16     # In common usage, we remove the field to let py-evm choose a
17     'timestamp': 1514764800,
18 }
```

- **100000**

```
11
12 GENESIS_PARAMS = {
13     'difficulty': 100000,
14     'gas_limit': 3141592,
15     # We set the timestamp, just to make this documented example r
16     # In common usage, we remove the field to let py-evm choose a
17     'timestamp': 1514764800,
18 }
```

```
**2** This is the result cap (PoW Nonce):  11579208923731619542357098500868790785326998466564056
Traceback (most recent call last):
  File "app/main.py", line 103, in <module>
    nonce, mix_hash = mine_pow_nonce(
  File "/home/checkout/python-blkchain/venv/lib/python3.8/site-packages/eth/consensus/pow.py",
    raise Exception("Too many attempts at POW mining, giving up")
Exception: Too many attempts at POW mining, giving up
(venv) checkout@boot-node:~/python-blkchain/demo-app$
```

- **A8: We can see the difficulty that is defined in the genesis block in the main.py file in the logs we have added when we are grabbing the new block header (screenshots contains this) and the difficulty is defined right there as a parameter of the block itself. The higher in number we went the longer the logs got as it was trying to figure out the PoW. The commands worked until the last one when we got an error from the Py-EVM and it tells us the stack trace to where it failed.**

- **Bonus:**
  - **B1:** Find a way to run the last difficulty from the last question. Provide your solution and the logs/code you implemented to make this work.

```
92
93
94 MAX_TEST_MINE_ATTEMPTS = 1000000
95
96
97 def mine_pow_nonce(block_number: int
```

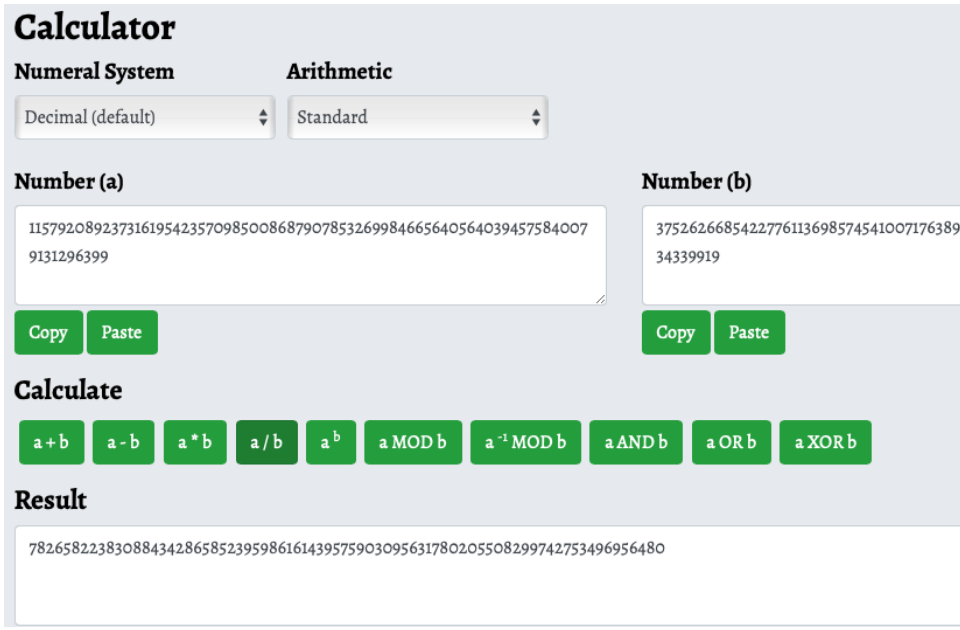
  - This can take an extremely long time. But this is how the real world operates. Of couse our VM is no where near as powerful as the purpose built miners, which is why if we were to implement a real world difficulty, the PoW would take hours

- **Note: At this point the lab was getting a little on the longer side as this is page 38/52. There is some more tasks we can look at in the lab exercises, they may be able to constitute another step, but would add another 10-15 pages in screenshots, analysis and writeup. They are covered in my Py-EVM playlist on YouTube documenting how the code works and what you can do with it. The youtube playlist is here: https://youtube.com/playlist?list=PLbYsGdagz-PJ1N2ocF6YFd76603ONLurg**
  - **The youtube playlist is around an hour long with 9 videos explaining what we can do with the code and the debugger environment.**
- **I have also included a full clone of my VM because there is a ton of setup work you need to do for the environment and application, and debugging code. There is a list of instructions to get you to running the code in a configured environment.**
- **On Startup of the VM**
  - Open the terminal and navigate to the “pythond-blkchain” folder.
  - Once there, “ls” you should be presented with the following
    - Chain
    - demo-app
    - sample_files
    - venv

- Enter in the following command to get to the virtual environment :

  . venv/bin/activate

- Your main application file is in the app folder, navigate to the demo-app folder. All the code has been uncommented so you can run the file right away and get the full logs. Follow the lab instructions starting in Exercise 3.
- To run the file enter the command: python3 app/main.py
- Your debugger should be up and running.



- **Appendicies**
  - **Appendix A:**
    - **q = 17**
    - **w = 10**

- **s = 20**
- **tot = (20 * 10) % q**
- **print(tot)**
-

- **Appendix B:**
  - from eth import constants
  - from eth.chains.mainnet import MainnetChain
  - from eth.db.atomic import AtomicDB
  - from eth_utils import to_wei, encode_hex

  - MOCK_ADDRESS = constants.ZERO_ADDRESS
  - DEFAULT_INITIAL_BALANCE = to_wei(10000, 'ether')

  - GENESIS_PARAMS = {

    'difficulty': constants.GENESIS_DIFFICULTY,

    }

  - GENESIS_STATE = {

    MOCK_ADDRESS: {

    "balance": DEFAULT_INITIAL_BALANCE,

```
        "nonce": 0,
        "code": b",
        "storage": {}
    }
}
```

- chain = MainnetChain.from_genesis(AtomicDB(), GENESIS_PARAMS, GENESIS_STATE)
- mock_address_balance = chain.get_vm().state.get_balance(MOCK_ADDRESS)
- print("The balance of address {} is {} wei".format(
  encode_hex(MOCK_ADDRESS),
  mock_address_balance)
-)

■ **Appendix C:**

- from eth_keys import keys
- from eth_utils import decode_hex
- from eth_typing import Address
- from eth import constants
- from eth.chains.base import MiningChain
- from eth.consensus.pow import mine_pow_nonce

- from eth.vm.forks.byzantium import ByzantiumVM
- from eth.db.atomic import AtomicDB

- GENESIS_PARAMS = {
  'difficulty': 1,
  'gas_limit': 3141592,
  # We set the timestamp, just to make this documented example reproducible.
  # In common usage, we remove the field to let py-evm choose a reasonable default.
  'timestamp': 1514764800,
  }

- SENDER_PRIVATE_KEY = keys.PrivateKey(
  decode_hex('0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8')
)

- SENDER = Address(SENDER_PRIVATE_KEY.public_key.to_canonical_address())

- RECEIVER =

  Address(b'\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\x02')

- klass = MiningChain.configure(

  __name__='TestChain',

  vm_configuration=(

  (constants.GENESIS_BLOCK_NUMBER, ByzantiumVM),

  ))

- chain = klass.from_genesis(AtomicDB(), GENESIS_PARAMS)
- genesis = chain.get_canonical_block_header_by_number(0)
- vm = chain.get_vm()
- nonce = vm.state.get_nonce(SENDER)
- tx = vm.create_unsigned_transaction(

  nonce=nonce,

  gas_price=0,

  gas=100000,

  to=RECEIVER,

  value=0,

  data=b'',

)

- signed_tx = tx.as_signed_transaction(SENDER_PRIVATE_KEY)

- chain.apply_transaction(signed_tx)

- block_result = chain.get_vm().finalize_block(chain.get_block())
- block = block_result.block
- nonce, mix_hash = mine_pow_nonce(

  block.number,

  block.header.mining_hash,

  block.header.difficulty

)
- print("this is the nonce: ", nonce)
- print("this is the mix_hash: ", mix_hash)
- chain.mine_block(mix_hash=mix_hash, nonce=nonce)

■ Appendix D: Simple_consensus.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 18 16:07:46 2022

@author: matthewmccreary
"""
import random
import time
```

```
# timer to show execution times
create_hash_start = time.time()

#open a file to write the output logs
#f = open("2-digit-brute.py.out.txt", "w")

# variables for the simple consensus algorithm.
target_hash = 1000005
nonce = 0

print("** Finding Target for Validation **")
yes = False
while yes != True:
    guess_hash = random.randint(1000000,9999999)
    f.write(str(guess_hash))
    f.write("\n")
    nonce += 1
    if guess_hash <= target_hash:
        yes = True
        f.close()

print("Target hash solved with: ", guess_hash)
```

```
print("Nonce : ", nonce)
f.close()
create_hash_end = time.time()




check_hash_start = time.time()
print("** Checking for Verification **")
if guess_hash <= target_hash:
    check_hash_end = time.time()
    print("Our supplied nonce was: ", nonce)
    print("Our found value is: ", guess_hash)
    print("The result is less than or equal to the target: ",
target_hash)
    # describe to the user what just happened


print("****************************************")
print("********** Performance Review **********")
print("****************************************")
print("Total execution time to create a valid hash: ",
(create_hash_end - create_hash_start))
```

```
print("Total execution time to check a valid hash: ",
(check_hash_end - check_hash_start))


print("\n ** Verification has been completed, this transaction has
been\n",
      "verified according to our consensus algorithm.  Or in other
terms\n",
      "there has been a consensus between at least two nodes that
this\n",
      "transaction and the supplied nonce are correct. **")
print("** To see process log, navigate to the 2-digit-brute.py.out.txt
file **")
```

Appendix E:

```
from eth_keys import keys
from eth_utils import decode_hex
from eth_typing import Address
from eth import constants
from eth.chains.base import MiningChain
from eth.consensus.pow import mine_pow_nonce
from eth.vm.forks.byzantium import ByzantiumVM
from eth.db.atomic import AtomicDB



GENESIS_PARAMS = {
    'difficulty': 100,
    'gas_limit': 3141592,
    # We set the timestamp, just to make this documented example reproducible.
    # In common usage, we remove the field to let py-evm choose a reasonable default.
    'timestamp': 1514764800,
```

```
}

SENDER_PRIVATE_KEY = keys.PrivateKey(
   decode_hex('0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8')
)
SENDER = Address(SENDER_PRIVATE_KEY.public_key.to_canonical_address())

RECEIVER = Address(b'\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\x02')

klass = MiningChain.configure(
   __name__='TestChain',
   vm_configuration=(
       (constants.GENESIS_BLOCK_NUMBER, ByzantiumVM),
   ))

chain = klass.from_genesis(AtomicDB(), GENESIS_PARAMS)
genesis = chain.get_canonical_block_header_by_number(0)
vm = chain.get_vm()



#print("**************************************************************************
*********")
#print("************************* Welcome to the py-evm logger
****************************")
#print("**************************************************************************
*********")
#print("Part 1")
#print("**************************************************************************
*********")
#print("We just created the chain, here it is: ", chain)
#print("We just created the mining chain object klass, here it is: ", klass)
#print("We just created the genesis block, here it is", genesis)
#print("We just created the VM, here it is: ", vm)




#this is the nonce from the sender, this if the nonce does not increase, then the
transaction will be considered queued, if the nonce increases, then the transaction
will be pending.  This is not the same nonce as the transaction nonce, this nonce is
tied to the account to track transactions.
```

```
nonce = vm.state.get_nonce(SENDER)



tx = vm.create_unsigned_transaction(
   nonce=nonce,
   gas_price=0,
   gas=100000,
   to=RECEIVER,
   value=0,
   data=b'',
)

signed_tx = tx.as_signed_transaction(SENDER_PRIVATE_KEY)

#print("**********************************************************************
*********")
#print("**********************************************************************
*********")
#print("**********************************************************************
*********")
#print("Part 2")
#print("**********************************************************************
*********")
#print("This is the the SENDER variable", SENDER)
#print("This is the result of the Address object for the receiver: ", RECEIVER)
##print("This is the sender's private key, this is it decoded", SENDER_PRIVATE_KEY)
#print("this is the unsigned transaction", tx)
#print("This is the user's nonce:  ",nonce)
#print("this is the signed transaction", signed_tx)




chain.apply_transaction(signed_tx)



# Normally, we can let the timestamp be chosen automatically, but
# for the sake of reproducing exactly the same block every time,
# we will set it manually here:
chain.set_header_timestamp(genesis.timestamp + 1)

# We have to finalize the block first in order to be able read the
```

```
# attributes that are important for the PoW algorithm
block_result = chain.get_vm().finalize_block(chain.get_block())
block = block_result.block


q = type(block_result)
w = type(block)


# based on mining_hash, block number and difficulty we can perform
# the actual Proof of Work (PoW) mechanism to mine the correct
# nonce and mix_hash for this block
nonce, mix_hash = mine_pow_nonce(
   block.number,
   block.header.mining_hash,
   block.header.difficulty
)
#print("**********************************************************************
*********")
#print("**********************************************************************
*********")
#print("**********************************************************************
*********")
#print("Part 3")
#print("**********************************************************************
*********")
#print("This is the block type result we got: ", q)
#print("This is the block type we got", w)
#print("This is the nonce value from the mine_pow_nonce method: ", nonce)
#print("This is the mix_hash value from the mine_pow_nonce method: ", mix_hash)

chain.mine_block(mix_hash=mix_hash, nonce=nonce)

#print("**********************************************************************
*********")
#print("**********************************************************************
*********")
#print("**********************************************************************
*********")
#print("Part 4")
#print("**********************************************************************
*********")
```

```
#print("We are mining the block now")
#print(block.number)
```

Appendix F:

- **Uncomment the following lines in according to the task you are on for Exercise 3**
- **When you have moved onto the next task comment out the task you just completed**
    - **Uncomment and leave uncommented**
        - **Lines: 39-41**
    - **T1: Blockchain Generation**
        - **Lines: 42-46**
    - **T2: Transaction Generation**
        - **Lines: 67-77**
    - **T3: Block Creation, Packing**
        - **Lines: 108-116**
    - **T4: Validation (Mining)**
        - **Lines: 120-126**
    - **T5: All**
        - **Uncomment all the lines above**
- **Appendix G**
    - **T1: validation.py**
        - **Print out the values “value” & “maximum” in the “def validate_let” function.**

- **T2: transactions.py**
  - **Print out the values "VRS((v,r,s))" & "transaction_parts"**
- **T3: base.py**
  - **Go to line 634, and add logging statements, here is the format I used for my logging, create your own with print statements, make sure they are easily identifiable in the output and provide relevant information**

```
634         vm = self.get_vm(self.header)
635         base_block = vm.get_block()
636         print("*************************************************************************")
637         print("*************************We are in the Apply Transaction Method**********************
638         print("*************************************************************************")
639         receipt, computation = vm.apply_transaction(base_block.header, transaction)
640         print("This is the receipt of applying a transaction", receipt)
641         print("This is the transaction that we applied", transaction)
642         header_with_receipt = vm.add_receipt_to_header(base_block.header, receipt)
643         print("This is the header with the receipt", header_with_receipt)
644         # since we are building the block locally, we have to persist all the incremental state
645         vm.state.persist()
646         new_header: BlockHeaderAPI = header_with_receipt.copy(state_root=vm.state.state_root)
647
648         transactions = base_block.transactions + (transaction, )
649         receipts = base_block.get_receipts(self.chaindb) + (receipt, )
650         print("This is after we make a new block header: ", transactions)
651         print("This is the block transaction receipts: ", receipts)
652         new_block = vm.set_block_transactions(base_block, new_header, transactions, receipts)
```

  - **If you preformed the above task correctly now navigate down to the "def mine_block_extended()" function. For me and the logs I created above put this on line 711.**

```
711     def mine_block_extended(self, *args: Any, **kwargs: Any) -> BlockAndMetaWitness:
712         custom_header = self._custom_header(self.header, **kwargs)
713         print("*************************************************************************")
714         print("*************************We now have a new header for a block**********************
715         print("*************************************************************************")
716         print("This is the custom header for the new block", custom_header)
717         print("*************************************************************************")
718         print("*************************We now are grabbing the current block**********************
719         print("*************************************************************************")
720         vm = self.get_vm(custom_header)
721         current_block = vm.get_block()
722         print("This is the current block in tuple format: ", tuple(current_block))
723         mine_result = vm.mine_block(current_block, *args, **kwargs)
724         print("This is the result of the vm.mine_block: ", mine_result)
725         mined_block = mine_result.block
726         print("This is our mined block", mined_block)
727         self.validate_block(mined_block)
728
```

- **T4: pow.py**

- **Like in the T3, you will need to create your own logs for this task, this is how I did mine, I added the 1 & 2 to show the flow internal of the file. Use the screenshots to guide you in where to find the methods to place the print commands in the pow.py file.**

```
65
66 def check_pow(block_number: int,
67                mining_hash: Hash32,
68                mix_hash: Hash32,
69                nonce: bytes,
70                difficulty: int) -> None:
71     validate_length(mix_hash, 32, title="Mix Hash")
72     validate_length(mining_hash, 32, title="Mining Hash")
73     validate_length(nonce, 8, title="POW Nonce")
74     cache = get_cache(block_number)
75     mining_output = hashimoto_light(
76         block_number, cache, mining_hash, big_endian_to_int(nonce))
77     print("**1** This is the output to the mining", mining_output)
78     if mining_output[b'mix digest'] != mix_hash:
79         raise ValidationError(
80             f"mix hash mismatch; expected: {encode_hex(mining_output[b'mix
81             f"!= actual: {encode_hex(mix_hash)}. "
82             f"Mix hash calculated from block #{block_number}, "
83             f"mine hash {encode_hex(mining_hash)}, nonce {encode_hex(nonce)
84             f", difficulty {difficulty}, cache hash {encode_hex(keccak(cach
85         )
86
87     result = big_endian_to_int(mining_output[b'result'])
88     print("**1** This is the result after the mining converted to int",resu
```

-

```
97 def mine_pow_nonce(block_number: int, mining_hash: Hash32, difficulty: int) -> Tuple[b
98     cache = get_cache(block_number)
99     for nonce in range(MAX_TEST_MINE_ATTEMPTS):
100         mining_output = hashimoto_light(block_number, cache, mining_hash, nonce)
101         result = big_endian_to_int(mining_output[b'result'])
102         result_cap = 2**256 // difficulty
103         print("**2** This is our result(PoW Nonce): ", result)
104         print("**2** This is the result cap (PoW Nonce): ", result_cap)
```

-