Stake auto-compounder

Introduction

The Stake Auto-Compounder is a CosmPy based use case developed using Python and designed to automate the process of staking tokens in a blockchain network, claiming rewards, and compounding those rewards by re-delegating them to a validator. When an account delegates tokens to a network's validator, it will start generating rewards proportionally to the amount ofStake ↗ delegated. But since rewards aren not automatically added to your stake and therefore do not contribute to future rewards, we can perform a compounding strategy to generate exponential rewards.

Delegate your tokens

The first thing we need to do is delegate some tokens to avalidator . You can do so by using aWallet ↗ and specifying the validator address and amount. you can delegate tokens to a specific validator by using thedelegate_tokens method of theledger_client object and specifying the validator's address, the amount of tokens and the wallet from which the delegation is made:

validators

ledger_client . query_validators ()

choose any validator

validator

validators [0]

key

PrivateKey ("FX5BZQcr+FNl2usnSIQYpXsGWvBxKLRDkieUNIvMOV7=") wallet =

LocalWallet (key)

delegate some tokens to this validator

tx

ledger_client . delegate_tokens (validator.address, 9000000000000000000 , wallet) tx . wait_to_complete ()

Auto-compounder

We can write a script helping us claiming rewards and delegating the rewarded tokens back to thevalidator of choice. This way we keep growing ourStake given the generated compounded rewards on such staked amount. We first need to define thetime limit and the compoundingperiod .

Importantly, bear in mind that each time an account performs a claim or a delegate a transaction, it has to pay certain fees. Therefore, the compounding period has to be long enough to generate sufficient rewards to exceed the fees that will be paid in each transaction and generate a profit.

After having defined such parameters, we can then start a timer that claims rewards and delegates them in each time period:

time_check

0 start_time = time . monotonic () time . sleep (period)

query, claim and delegate rewards after time period

while time_check < time_limit :

begin

time . monotonic ()

summary

ledger_client . query_staking_summary (wallet. address ()) print (f "Staked: { summary.total_staked } ")

balance_before

ledger_client . query_bank_balance (wallet. address ())

tx

ledger_client . claim_rewards (validator.address, wallet) tx . wait_to_complete ()

balance_after

ledger_client . query_bank_balance (wallet. address ())

reward after any fees

true_reward

balance_after

balance_before

if true_reward

0 :

print (f "Staking { true_reward } (reward after fees)")

tx

ledger_client . delegate_tokens (validator.address, true_reward, wallet) tx . wait_to_complete ()

else : print ("Fees from claim rewards transaction exceeded reward")

end

time . monotonic ()

time . sleep (period - (end - begin)) time_check = time . monotonic ()

- start_time In the code snippet above we defined a while loop running until the timer exceeds thetime limit . Each loop will last the time specified inperiod . We query the balance before and after claiming rewards to get the value of the reward after any fees. If the true reward value is positive, we delegate those tokens to the validator, if it is negative, it means that the fees from claiming and delegating transactions exceeded the rewards, and therefore we will not delegate.

Walk-through

Below we provide a step-by-step guide to create an auto compounder using thecosmpy.aerial package.

1. First of all, create a Python script and name it:touch aerial_compounder.py

2. We need to import the necessary modules, includingargparse
3. ,time
4. , and various modules from thecosmpy.aerial
5. package:
6. import
7. argparse
8. import
9. time
10. from
11. cosmpy
12. .
13. aerial
14. .
15. client
16. import
17. LedgerClient
18. from
19. cosmpy
20. .
21. aerial
22. .
23. config
24. import
25. NetworkConfig
26. from
27. cosmpy
28. .
29. aerial
30. .
31. faucet
32. import
33. FaucetApi
34. from
35. cosmpy
36. .
37. aerial
38. .
39. wallet
40. import
41. LocalWallet
42. We now need to define a_parse_commandline()
43. function responsible for parsing command-line arguments when the script is being executed:
44. def
45. _parse_commandline
46. ():
47. parser
48. =
49. argparse
50. .
51. ArgumentParser
52. ()
53. parser
54. .
55. add_argument
56. (
57. "initial_stake"
58. ,
59. type
60. =
61. int
62. ,
63. nargs
64. =
65. "?"
66. ,
67. default
68. =
69. 9000000000000000000

,
help
=
"Initial amount of atestfet to delegate to validator"
,
)
parser
.
add_argument
(
"time_limit"
,
type
=
int
,
nargs
=
"?"
,
default
=
600
,
help
=
"total time"
,
)
parser
.
add_argument
(
"period"
,
type
=
int
,
nargs
=
"?"
,
default
=
100
,
help
=
"compounding period"
,
)
return
parser
.
parse_args
()
We first create aparser
instance of theArgumentParser
class using theargparse
module. Argument parsers are used to specify and parse command-line arguments. Theadd_argument()
method is used to specify the arguments that the script will accept. It takes several parameters, including:

- name
- : the name of the argument.
- type

- : the type to which the argument should be converted (in this case,int
-).
- nargs
- : the number of arguments expected (in this case,"?"
- means zero or one argument).
- default
- : the default value if the argument is not provided.
- help
- : a brief description of the argument, which will be displayed if the user asks for help with the script.

Three arguments are defined in this function:

- initial_stake
- : the initial amount of tokens to delegate to a validator. It expects an integer and has a default value of9000000000000000000
- .
- time_limit
- : the total time limit for the compounder. It expects an integer (representing seconds) and has a default value of600
- seconds (10 minutes).
- period
- : the compounding period, which is the interval between each compounding operation. It expects an integer (also in seconds) and has a default value of100
- seconds.

The last line of the snippet above,parser.parse_args()
, parses the command-line arguments provided when the script is executed. The function returns the parsed arguments object.
We are now ready to define ourmain()
function:

```
def
main
():
"""Run main."""
args
=
_parse_commandline
()
ledger
=
LedgerClient
(NetworkConfig.
fetchai_stable_testnet
())
faucet_api
=
FaucetApi
(NetworkConfig.
fetchai_stable_testnet
())
```

get all the active validators on the network

```
validators
=
ledger
.
query_validators
()
```

choose any validator

```
validator
=
validators
[
0
]
alice
=
LocalWallet
.
generate
()
wallet_balance
=
ledger
.
query_bank_balance
(alice.
address
())
initial_stake
=
args
.
initial_stake
while
wallet_balance
<
(initial_stake)
:
print
(
"Providing wealth to wallet..."
)
faucet_api
.
get_wealth
(alice.
address
())
wallet_balance
=
ledger
.
query_bank_balance
(alice.
address
())
```

delegate some tokens to this validator

```
tx
=
ledger
.
delegate_tokens
(validator.address, initial_stake, alice)
```

tx
.
wait_to_complete
()

set time limit and compounding period in seconds

time_limit
=
args
.
time_limit
period
=
args
.
period
time_check
=
0
start_time
=
time
.
monotonic
()
time
.
sleep
(period)

query, claim and stake rewards after time period

while
time_check
<
time_limit
:
begin
=
time
.
monotonic
()
summary
=
ledger
.
query_staking_summary
(alice.
address
())
print
(
f
"Staked:
{
summary.total_staked
}
"
)
balance_before
=
ledger
.
query_bank_balance

(alice.
address
())
tx
=
ledger
.
claim_rewards
(validator.address, alice)
tx
.
wait_to_complete
()
balance_after
=
ledger
.
query_bank_balance
(alice.
address
())

reward after any fees

true_reward
=
balance_after
-
balance_before
if
true_reward

0
:
print
(
f
"Staking
{
true_reward
}
(reward after fees)"
)
tx
=
ledger
.
delegate_tokens
(validator.address, true_reward, alice)
tx
.
wait_to_complete
()
else
:
print
(
"Fees from claim rewards transaction exceeded reward"
)
print
()
end
=
time
.
monotonic

367. ()
368. time
369. .
370. sleep
371. (period
372. -
373. (end
374. -
375. begin))
376. time_check
377. =
378. time
379. .
380. monotonic
381. ()
382. -
383. start_time
384. if
385. **name**
386. ==
387. "**main**"
388. :
389. main
390. ()
391. The first line calls the_parse_commandline()
392. function we defined earlier. It returns an object with the parsed command-line arguments. We then create two objects:
393.
    - Aledger
394.
    - instance of theLedger Client
395.
    - class configured for the Fetch.ai stable testnet. This client will be used to interact with the blockchain network.
396.
    - Afaucet_api
397.
    - instance of theFaucet API
398.
    - class configured for the Fetch.ai stable testnet. This API is used for providing additional funds to the wallet if needed.
399. We then need to get all the active validators on the network by using thequery_validators()
400. method. After this, we choose a validator and create a new wallet namedalice
401. usingLocalWallet.generate()
402. and check the balance of thealice
403. wallet. If the balance is less than the initial stake, it enters a loop to provide wealth to the wallet using the faucet API until the balance reaches the specified initial stake. We can now delegate the initial stake of tokens to the chosen validator using thedelegate_tokens()
404. method.
405. We proceed by setting time limits and periods.time_limit = args.time_limit
406. sets the time limit based on the command-line argument, whereasperiod = args.period
407. sets the compounding period based on the command-line argument. After this, we define the compounding loop, similar to what was described in the first part of this guide: it iterates over a specified time period, queries staking summary, claims rewards, and either stakes the rewards or skips if fees exceed rewards. Time management is important here: indeed, the loop keeps track of time usingtime.monotonic()
408. to ensure it does not exceed the specified time limit. It waits for the specified period before starting the next compounding cycle.
409. Save the script.

The overall script should look as follows:

aerial_compounder.py import argparse import time

from cosmpy . aerial . client import LedgerClient from cosmpy . aerial . config import NetworkConfig from cosmpy . aerial . faucet import FaucetApi from cosmpy . aerial . wallet import LocalWallet

def

_parse_commandline (): parser = argparse . ArgumentParser () parser . add_argument ("initial_stake" , type = int , nargs = "?" , default = 9000000000000000000 , help = "Initial amount of atestfet to delegate to validator" ,) parser . add_argument ("time_limit" , type = int , nargs = "?" , default = 600 , help = "total time" ,) parser . add_argument ("period" , type = int ,

nargs = "?" , default = 100 , help = "compounding period" ,)

return parser . parse_args ()

def

main (): """Run main."""" args =

_parse_commandline ()

ledger

LedgerClient (NetworkConfig. fetchai_stable_testnet ()) faucet_api =

FaucetApi (NetworkConfig. fetchai_stable_testnet ())

get all the active validators on the network

validators

ledger . query_validators ()

choose any validator

validator

validators [0]

alice

LocalWallet . generate ()

wallet_balance

ledger . query_bank_balance (alice. address ()) initial_stake = args . initial_stake

while wallet_balance < (initial_stake) : print ("Providing wealth to wallet...") faucet_api . get_wealth (alice. address ()) wallet_balance = ledger . query_bank_balance (alice. address ())

delegate some tokens to this validator

tx

ledger . delegate_tokens (validator.address, initial_stake, alice) tx . wait_to_complete ()

set time limit and compounding period in seconds

time_limit

args . time_limit period = args . period

time_check

0 start_time = time . monotonic () time . sleep (period)

query, claim and stake rewards after time period

while time_check < time_limit :

begin

time . monotonic ()

summary

ledger . query_staking_summary (alice. address ()) print (f "Staked: { summary.total_staked } ")

balance_before

ledger . query_bank_balance (alice. address ())

tx

ledger . claim_rewards (validator.address, alice) tx . wait_to_complete ()

balance_after

ledger . query_bank_balance (alice. address ())

reward after any fees

true_reward

balance_after

balance_before

if true_reward

0 :

print (f "Staking { true_reward } (reward after fees)")

tx

ledger . delegate_tokens (validator.address, true_reward, alice) tx . wait_to_complete ()

else : print ("Fees from claim rewards transaction exceeded reward")

print ()

end

time . monotonic () time . sleep (period - (end - begin)) time_check = time . monotonic ()

- start_time

if

name

==

"**main**" : main ()

Was this page helpful?

Smart contracts Stake optimizer