

# 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 of [Stake ↗](#) delegated. But since rewards aren't 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 a validator. You can do so by using [a Wallet ↗](#) and specifying the validator address and amount. You can delegate tokens to a specific validator by using the `delegate_tokens` method of the `ledger_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+FNI2usnSIQYpXsGWvBxKLRDkieUNlvMOV7=" ) 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 the validator of choice. This way we keep growing our stake given the generated compounded rewards on such staked amount. We first need to define the time limit and the compounding period.

Importantly, bear in mind that each time an account performs a claim or a delegate 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 the time limit . Each loop will last the time specified in period . 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 the `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, including argparse
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

```

```

70. ,
71. help
72. =
73. "Initial amount of atestfet to delegate to validator"
74. ,
75. )
76. parser
77. .
78. add_argument
79. (
80. "time_limit"
81. ,
82. type
83. =
84. int
85. ,
86. nargs
87. =
88. "?"
89. ,
90. default
91. =
92. 600
93. ,
94. help
95. =
96. "total time"
97. ,
98. )
99. parser
100. .
101. add_argument
102. (
103. "period"
104. ,
105. type
106. =
107. int
108. ,
109. nargs
110. =
111. "?"
112. ,
113. default
114. =
115. 100
116. ,
117. help
118. =
119. "compounding period"
120. ,
121. )
122. return
123. parser
124. .
125. parse_args
126. ()
127. We first create a parser
128. instance of the ArgumentParser
129. class using the argparse
130. module. Argument parsers are used to specify and parse command-line arguments. The add_argument()
131. method is used to specify the arguments that the script will accept. It takes several parameters, including:
132.
    ◦ name
133.
    ◦ : the name of the argument.
134.
    ◦ type

```

```

135.     ◦ : the type to which the argument should be converted (in this case,int
136.     ◦ ).
137.     ◦ nargs
138.     ◦ : the number of arguments expected (in this case,"?"
139.     ◦ means zero or one argument).
140.     ◦ default
141.     ◦ : the default value if the argument is not provided.
142.     ◦ help
143.     ◦ : a brief description of the argument, which will be displayed if the user asks for help with the script.
144. Three arguments are defined in this function:
145.     ◦ initial_stake
146.     ◦ : the initial amount of tokens to delegate to a validator. It expects an integer and has a default value
        of9000000000000000000
147.     ◦ .
148.     ◦ time_limit
149.     ◦ : the total time limit for the compounder. It expects an integer (representing seconds) and has a default value
        of600
150.     ◦ seconds (10 minutes).
151.     ◦ period
152.     ◦ : the compounding period, which is the interval between each compounding operation. It expects an integer (also
        in seconds) and has a default value of100
153.     ◦ seconds.
154. The last line of the snippet above,parser.parse_args()
155. , parses the command-line arguments provided when the script is executed. The function returns the parsed
    arguments object.
156. We are now ready to define ourmain()
157. function:
158. def
159. main
160. ():
161. """Run main."""
162. args
163. =
164. _parse_commandline
165. ()
166. ledger
167. =
168. LedgerClient
169. (NetworkConfig.
170. fetchai_stable_testnet
171. ())
172. faucet_api
173. =
174. FaucetApi
175. (NetworkConfig.
176. fetchai_stable_testnet
177. ())

```

178. **get all the active validators on the network**

```
179. validators
180. =
181. ledger
182. .
183. query_validators
184. ()
```

## 185. **choose any validator**

```
186. validator
187. =
188. validators
189. [
190. 0
191. ]
192. alice
193. =
194. LocalWallet
195. .
196. generate
197. ()
198. wallet_balance
199. =
200. ledger
201. .
202. query_bank_balance
203. (alice.
204. address
205. ())
206. initial_stake
207. =
208. args
209. .
210. initial_stake
211. while
212. wallet_balance
213. <
214. (initial_stake)
215. :
216. print
217. (
218. "Providing wealth to wallet..."
219. )
220. faucet_api
221. .
222. get_wealth
223. (alice.
224. address
225. ())
226. wallet_balance
227. =
228. ledger
229. .
230. query_bank_balance
231. (alice.
232. address
233. ())
```

## 234. **delegate some tokens to this validator**

```
235. tx
236. =
237. ledger
238. .
239. delegate_tokens
240. (validator.address, initial_stake, alice)
```

```
241. tx
242. .
243. wait_to_complete
244. ()
```

## 245. **set time limit and compounding period in seconds**

```
246. time_limit
247. =
248. args
249. .
250. time_limit
251. period
252. =
253. args
254. .
255. period
256. time_check
257. =
258. 0
259. start_time
260. =
261. time
262. .
263. monotonic
264. ()
265. time
266. .
267. sleep
268. (period)
```

## 269. **query, claim and stake rewards after time period**

```
270. while
271. time_check
272. <
273. time_limit
274. :
275. begin
276. =
277. time
278. .
279. monotonic
280. ()
281. summary
282. =
283. ledger
284. .
285. query_staking_summary
286. (alice.
287. address
288. ())
289. print
290. (
291. f
292. "Staked:
293. {
294. summary.total_staked
295. }
296. "
297. )
298. balance_before
299. =
300. ledger
301. .
302. query_bank_balance
```

```
303. (alice.  
304. address  
305. ())  
306. tx  
307. =  
308. ledger  
309. .  
310. claim_rewards  
311. (validator.address, alice)  
312. tx  
313. .  
314. wait_to_complete  
315. ()  
316. balance_after  
317. =  
318. ledger  
319. .  
320. query_bank_balance  
321. (alice.  
322. address  
323. ())
```

## 324. **reward after any fees**

```
325. true_reward  
326. =  
327. balance_after  
328. -  
329. balance_before  
330. if  
331. true_reward  
332.  
  
333. 0  
334. :  
335. print  
336. (  
337. f  
338. "Staking  
339. {  
340. true_reward  
341. }  
342. (reward after fees)"  
343. )  
344. tx  
345. =  
346. ledger  
347. .  
348. delegate_tokens  
349. (validator.address, true_reward, alice)  
350. tx  
351. .  
352. wait_to_complete  
353. ()  
354. else  
355. :  
356. print  
357. (  
358. "Fees from claim rewards transaction exceeded reward"  
359. )  
360. print  
361. ()  
362. end  
363. =  
364. time  
365. .  
366. 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](#)