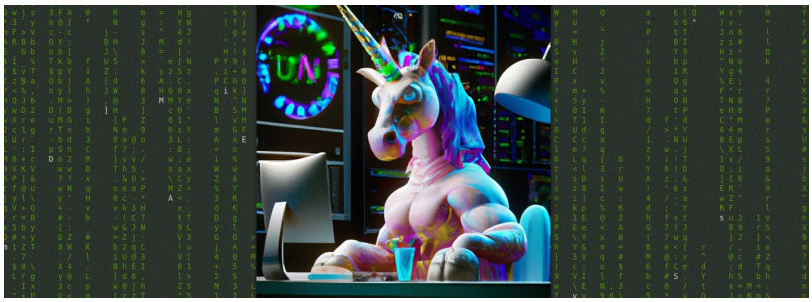
@elnaril
Elnaril

How to Buy a Token on the Uniswap Universal Router with Python



This tutorial will teach you how to use Python to build the Ethereum transaction you need to send to the Uniswap Universal Router (UR) in order to buy or swap tokens. Since there is no better learning method than practising on an actual project, I will walk you through a simple one: buy some UNI with 1 ETH.
:::info By the end of this tutorial, you will be able to create a transaction to buy some tokens from the latest Uniswap router with Python.
:::

Prerequisites

This tutorial is intended for developers who are comfortable with Python and the basics of Ethereum (contract, ERC20, transactions, gas, …). Already knowing the web3.py library is recommended but not strictly mandatory for this tutorial. Being familiar with the previous Uniswap routers or at least with the concept of automated market maker would help but is not necessary.

You will need the address of a RPC endpoint to request the blockchain. If you don't have a direct access to a node, you can get a free one from several providers: Infura, QuickNode, or Alchemy for examples.

At the moment of writing, the libraries we are going to use support Python 3.8 to 3.11, so be sure to have one of these versions installed on your computer, along with pip, the standard package manager.

Libraries and Tools

Here is the list of libraries and tools we are going to use in this tutorial. I'll show you how to install and use them to achieve our goal.

Python libraries

Web3.py
The primary library for Python developers who work with Ethereum compatible blockchains

UR Codec
The open source library that encodes and decodes the data sent to the Universal Router.

Tools

:::tip You don't want to learn to swap tokens with actual ones. So you can use either a testnet like Sepolia or a local fork created with Ganache which is the solution I'll use in this tutorial.
:::

Ganache CLI
We'll create a local fork of Ethereum at a given block number so you can get exactly the same results as me.

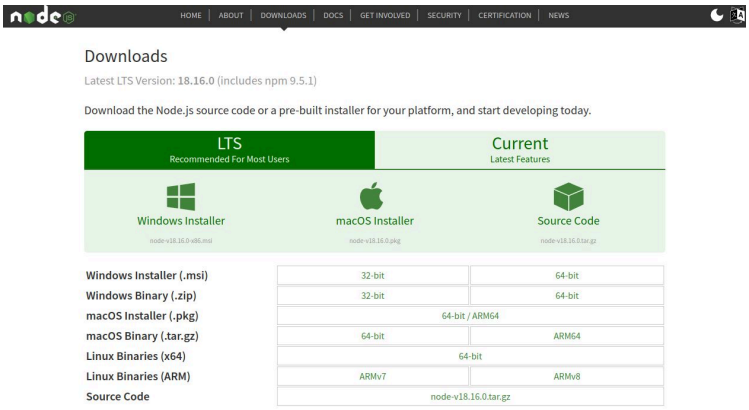
Node.js and npm
The JavaScript runtime environment used to run Ganache, and its package manager.

Setup

So, now that we have set the scene, let's get our hands dirty!
:::info By the end of this section, you'll have the tools and libraries installed and ready to use. You will also generate a private key which is mandatory to sign transactions.
:::

1. Node.js and npm installation

Go to the official download page and install Node.js accordingly to your OS.



Installing Node.js should be rather straightforward, but here is a great installation guide for Linux users.

At this point, you should be able to get something similar to:

```
$ node -v
v18.16.0
$ npm version
{
npm: '9.6.2',
node: '18.16.0',
...
}
```

2. Ganache installation

Once Node.js is installed, installing Ganache is as simple as:

```
$ npm install ganache --global
```

3. Launching Ganache

We're going to launch Ganache and instruct it to locally fork Ethereum at block number "17365005". You can use whatever mined block you wish or the latest one (remove the option in this case), but using the same block number will allow you to get the same results as me.

You will also need the address of your RPC endpoint. If you use Infura, that would be something like:

```
https://mainnet.infura.io/v3/xxx
```

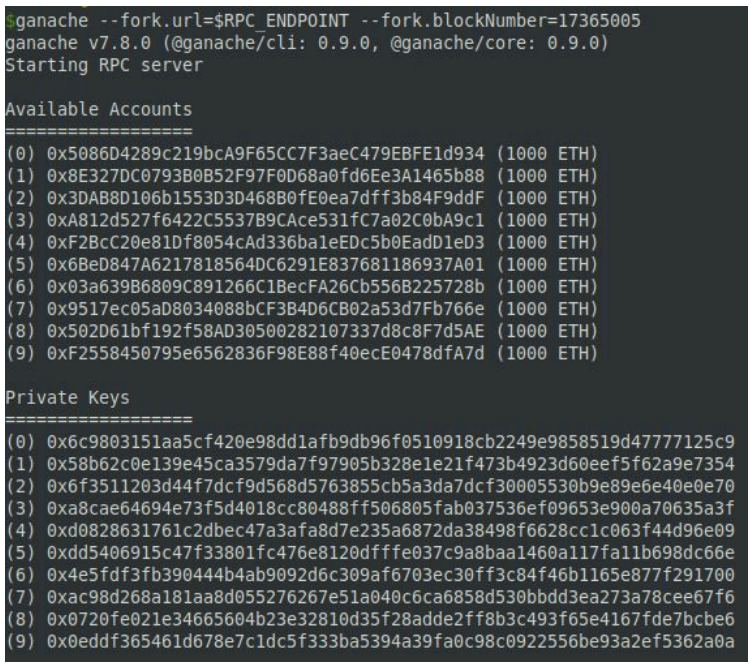
where xxx is your API key.

Mine is stored in the environment variable `$RPC_ENDPOINT`, so the command to launch Ganache would be:

```
ganache --fork.url=$RPC_ENDPOINT --
fork.blockNumber=17365005
```

4. Getting a private key from Ganache

When Ganache starts, it outputs several useful information which should start with something similar to:



As you can see, it generated 10 accounts (= 10 public/private key pairs) for you and fills them up with 1000 ETH! No need to rush to Etherscan to contemplate your new wealth! :joy: These ethers are only on your local fork!

Keep one of the private key for later usage. I'll keep this one:

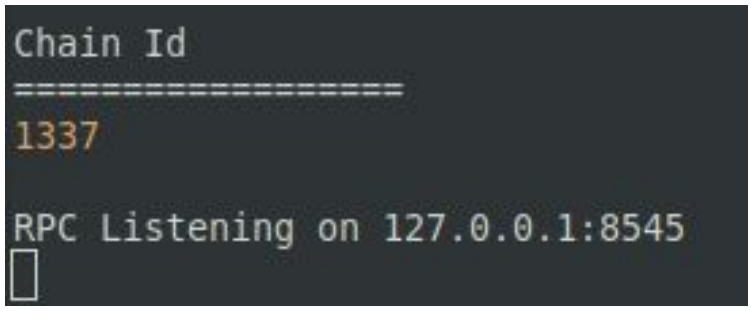
```
private_key = "0x6c9803151aa5cf420e98dd1afb9db96f0510918cb2249e9858519d47777125c9"
```

\

:::warning Do **not** use these private keys/accounts on Mainnet! Everyone could access your fund!
Also, if you see some tokens in these accounts on Mainnet (or in any account whose private key is on the web), **never** try to get them: a sweeper bot will steal your gas!
:::

5. Local chain id and rpc endpoint

The last part of what Ganache displayed at startup concerns the local chain id and the local rpc address.



We'll use them in our Python code.

```
chain_id = 1337
rpc_endpoint = "http://127.0.0.1:8545"
```

\

3. Python libraries installation

Now that Ganache is installed and running, let's do what we, Python developers, love to do: pythonic stuff! :laughing:

3.a. Python Virtual Environment

And we start with a good practice which is to use Python virtual environments. They are great to prevent dependency version conflicts, between all your projects and between them and your OS if it uses some Python. For those who are not familiar with virtual environments, here is a tutorial.

If you really don't want to bother with using a virtual environment right now, you can skip this part and go directly to the next sub-section.

First create a folder for your project and navigate into it. Then, creating a new virtual environment can be done with the following command:

```
$ python3.9 -m venv venv_ur_tutorial
```

Using Python 3.9, we used the built-in module `venv` (you could also use virtualenv) to create a virtual environment named "venv_ur_tutorial" that we can activate like this for Linux or Mac users:

```
$ source venv_ur_tutorial/bin/activate
```

or for Windows users:

```
venv_ur_tutorial/Scripts/activate.bat
```

At this step, your virtual environment is activated. It means that what ever dependency you install from this terminal/environment will only be "visible" within it.

3.b. Dependency installation

Now we'll install the latest pip version and the blockchain libraries:

```
$ pip install -U pip
$ pip install web3 uniswap-universal-router-decoder
```

To check you have the correct dependency versions, use the following command:

```
$ pip freeze
```

It will display all installed packages in the virtual environment. Just check it outputs these ones:

```
uniswap-universal-router-decoder==0.8.0
web3==6.4.0
```

That’s it for the installation! You have everything to continue to the most exciting parts: gathering the information needed for the swap and coding the transaction!

Gather the information we need

:::info By the end of this section, you will have all the data needed to build a swap transaction.
:::

1. From CoinGecko

I’ll use the data aggregator CoinGecko, but you can use CMC or whatever data provider you prefer.

A quick look on the CoinGecko page for the Uniswap token gives us the following information:

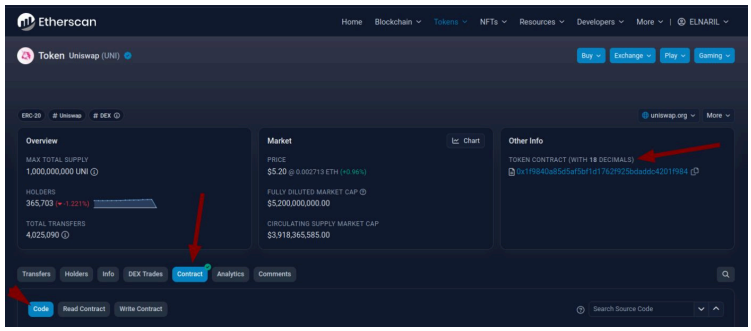
- The token is valued around 0.002715 ETH, so 1 ETH would be ~368 UNI
- Its address on Ethereum is `0x1f9840a85d5af5bf1d1762f925bdaddc4201f984`
- Its Etherscan link is https://etherscan.io/token/0x1f9840a85d5af5bf1d1762f925bdaddc4201f984

```
uni_address = Web3.to_checksum_address('0x1f9840a85d5af5bf1d1762f925bdaddc4201f984')
```

Web3 works only with checksum address, so we convert the string into the correct format thanks to the `Web3.to_checksum_address()`

2. From Etherscan

Follow the Etherscan link.



It tells you the token number of decimals: 18

Now click on the “Contract” tab, scroll down and copy the contract ABI.


```
uni_abi = '[{"inputs":[{"internalType":"address", ... ,"type":"function"}]'
```

We’ll use it to request the UNI contract. For readability sake, I’ve truncated most of it. If you have a doubt, you can check the full tutorial code.

3. Amounts

Using some of the information collected above, let’s calculate the amounts that will be swapped:

ETH:
In this project, we want to buy for 1 ETH of UNI. The Ethereum native coin is defined with 18 decimals. So the amount of ETH we want to pay (input in the swap) is:

```
amount_in = 1 * 10**18
```

UNI:
We saw that the current value for 1 ETH is about 368 UNI.

This price is an aggregation from multiple sources and considering market fluctuations and a possible slippage (well, not really on our local fork, but that could be the case on Mainnet), let’s give a bit of room to the amount we’re happy to receive in order to be sure the swap is successful. Let’s say we don’t want to receive less than 365 UNI for our 1 ETH.

The UNI token is also defined with 18 decimals. So the minimum amount of UNI we’re ready to receive out of the swap is:

```
min_amount_out = 365 * 10**18
```

Path

:::tip To find automatically the paths, more advanced developers can use the Smart Path open source library
:::
We need to instruct the Uniswap protocol on what tokens to swap. ETH is the native coin, but only tokens can be swapped, so we'll use the wrapped version which is WETH.

As an exercise, I let you find its address:

```
weth_address = Web3.to_checksum_address('0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2')
```

The path is then simply the list formed by the address of the token we provide and the one we want:

```
path = [weth_address, uni_address]
```

:::info This path is for a Uniswap V2 pool. V3 pools are slightly more complicated and are out of scope for this tutorial.
:::
:::info V2 paths can be formed with 3 token addresses when there is no pool for the 2 tokens we swap, which is not the case here.
:::

4. Universal Router Address and ABI

Lastly, the UR address is:

```
ur_address = Web3.to_checksum_address("0xEf1c6E67703c7BD7107eed8303Fbe6EC2554BF6B")
```

and the UR contract ABI is:

```
ur_abi = '[{"inputs":[{"components":[{"internalType": ... ,"type":"receive"}]'
```

For readability sake, I've truncated most of it. If you have a doubt, you can check the full tutorial code.

Coding a Transaction to Buy some UNI from the Uniswap Universal Router

:::info By the end of this section, you will know how to create and send a transaction to the UR with Python to buy some tokens.
:::

1. Creating a Python script

Now that we have the information we need, let's put them together in a python file that we call `buy_token.py`. We'll start by importing the libraries as follow:

```
from uniswap_universal_router_decoder import FunctionRecipient, RouterCodec
from web3 import Account, Web3
```

"**RouterCodec**" is the class used to encode and decode the data sent to the UR.
"**FunctionRecipient**" is an Enum that will be used to tell UR functions who is the receiver. It is often either the router itself or the transaction's sender.
"**Account**" is the class used to create an account and sign transactions.
"**Web3**" is the class used to interact with blockchains.

Then we add the constants we have defined in the previous sections:

```
from uniswap_universal_router_decoder import FunctionRecipient, RouterCodec
from web3 import Account, Web3
private_key = "0x6c9803151aa5cf420e98dd1afb9db96f0510918cb2249e9858519d47777125c9"
chain_id = 1337
rpc_endpoint = "http://127.0.0.1:8545"
uni_address = Web3.to_checksum_address('0x1f9840a85d5af5bf1d1762f925bdaddc4201f984')
uni_abi = '[{"inputs":[{"internalType":"address", ... ,"type":"function"}]'
amount_in = 1 * 10**18
min_amount_out = 365 * 10**18
weth_address = Web3.to_checksum_address('0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2')
path = [weth_address, uni_address]
ur_address = Web3.to_checksum_address("0xEf1c6E67703c7BD7107eed8303Fbe6EC2554BF6B")
ur_abi = '[{"inputs":[{"components":[{"internalType": ... ,"type":"receive"}]'
```

:::tip Be sure to replace the value of `uni_abi` and `ur_abi` with their correct value. If you're not sure, have look into the tutorial full code.
:::

2. Encoding the transaction input data

The input data is the part of the transaction that will be executed by the UR smart contract, resulting in the actual swap. We'll use the UR codec as follow:

First we instantiate it:

```
codec = RouterCodec()
```

Then we use it to encode the input data:

```
encoded_input = (
codec
.encode
.chain()
.wrap_eth(FunctionRecipient.ROUTER, amount_in)
.v2_swap_exact_in(FunctionRecipient.SENDER, amount_in, min_amount_out, path, payer_is_sender=False)
.build(codec.get_default_deadline())
)
```

Ok, but what does it mean ?! What does it do ?!

3. Some explanations

Let’s breakdown the command:

`encode` : tells the codec we want to encode an input data (as opposed to decode).

`chain()` : the UR supports several commands chained in a single transaction.
`chain()` initialises the chaining for one or more sub-commands.

`wrap_eth()` : the first sub-commands. It asks the router to convert `amount_in` ETH into WETH.
`FunctionRecipient.ROUTER` : the router will receive the WETH converted by `wrap_eth()`.

`v2_swap_exact_in()` : Instruct the router to use a V2 pool with known input amount (1 WETH in this case).
`FunctionRecipient.SENDER` : the transaction’s sender will receive the output of `v2_swap_exact_in()` (so the UNI tokens).
`payer_is_sender=False` : The `amount_in` UNI received by `v2_swap_exact_in()` will not come from the sender. This is because the corresponding UR function receives the WETH generated previously and kept by the router.
`min_amount_out`: If the swap results in less that this amount of UNI, the transaction will be reverted.
`path` : the tokens involved in the swap.

`codec.get_default_deadline()` : the timestamp after which the transaction will not be valid anymore.
`build()` : This method build and encode the transaction input data.

4. Build the transaction dictionary

First we need a `Web3` instance so we can interact with the blockchain:

```
w3 = Web3(Web3.HTTPProvider(rpc_endpoint))
```

and a wallet/account using the private key:

```
account = Account.from_key(private_key)
```

And now we're ready to create the transaction dictionary:

```
trx_params = {
"from": account.address,
"to": ur_address,
"gas": 500_000,
"maxPriorityFeePerGas": w3.eth.max_priority_fee,
"maxFeePerGas": 100 * 10**9,
"type": '0x2',
"chainId": chain_id,
"value": amount_in,
"nonce": w3.eth.get_transaction_count(account.address),
"data": encoded_input,
}
```

The code is pretty much self-explanatory here.
"`gas`" is the maximum amount of gas you're willing to pay. Everything not used will be refunded. Here `500_000` is totally arbitrary.
"`maxFeePerGas`": maximum gas price you're willing to pay. Here 100 Gwei is totally arbitrary.
:::tip Out of scope here, but you may want to have a more subtle gas / maxPriorityFeePerGas / maxFeePerGas computation!
:::
"`nonce`": a counter used to distinguish transactions sent from the same account.
"`data`" : contain the `encoded_input` previously computed.

5. Sign and send the transaction

Signing and sending the transaction is as simple as:

```
raw_transaction = w3.eth.account.sign_transaction(trx_params, account.key).rawTransaction
trx_hash = w3.eth.send_raw_transaction(raw_transaction)
```

That's it ! You have bought some UNI tokens with 1 ETH!

Bonus: let's check how many UNI we got

We need first to create a Web3 instance of the UNI contract from its ABI and address:

```
uni_contract = w3.eth.contract(address=uni_address, abi=uni_abi)
```

and call it to get our UNI balance:

```
uni_balance = uni_contract.functions.balanceOf(account.address).call()
print(uni_balance / 10**18)
```

My result is:

```
368.41333521045374
```

:::info If you made it up to here, congrats !!!
You have learnt:

- how to set up a Ganache environment
- what Python libraries to use
- Encode, build and send a swap to the Uniswap Universal Router

:::

Closing words and disclaimer

You can find the full code of this tutorial here.

Now that you know the basics of the UR codec, you can continue to play with the other functions it supports.

Disclaimer: I'm the author and maintainer of the UR codec. This open source library is a work in progress and does not support all UR functions yet. If you ever experience any issue with it, please open a ticket here

Tags: python,cryptocurrency,defi,software-development,uniswap,programming,ethereum,blockchain

Published On: Mon Jun 05 2023 09:47:31 GMT+0000 (Coordinated Universal Time)!

View Story Live at: https://hackernoon.com/how-to-buy-a-token-on-the-uniswap-universal-router-with-python!