Polaris project

Table of Contents

[2. Global project organisation & structure 4](#_Toc83643485)

[2.1. Main blocks 4](#_Toc83643486)

[3. Session & Session Manager 5](#_Toc83643487)

[3.1. Socket callbacks 5](#_Toc83643488)

[3.1.1. From Binance to ClientManager class 5](#_Toc83643489)

[3.1.2. From ClientManager class to MarketSocketsIn class 5](#_Toc83643490)

[3.1.3. From MarketSocketsIn class to SessionManager class 5](#_Toc83643491)

[3.2. Session 5](#_Toc83643492)

[3.2.1. Criteria to allow new perfect trade creation 6](#_Toc83643493)

[4. Client & Client Manager 7](#_Toc83643494)

[5. Dashboard 8](#_Toc83643495)

[6. Binance API 9](#_Toc83643496)

[6.1. Exchange info 9](#_Toc83643497)

[6.2. Get symbol info 11](#_Toc83643498)

[6.3. Place order 11](#_Toc83643499)

[6.4. Cancel order 12](#_Toc83643500)

[6.5. Order traded 13](#_Toc83643501)

[6.6. Websockets 14](#_Toc83643502)

[6.7. User socket 16](#_Toc83643503)

[6.8. API errors 22](#_Toc83643504)

[6.9. Messages for -1010 ERROR\_MSG\_RECEIVED, -2010 NEW\_ORDER\_REJECTED, and -2011 CANCEL\_REJECTED 22](#_Toc83643505)

[6.10. 23](#_Toc83643506)

[6.11. -9xxx Filter failures 23](#_Toc83643507)

[7. Python misc 25](#_Toc83643508)

[7.1. Command line arguments 25](#_Toc83643509)

[7.2. Exceptions 25](#_Toc83643510)

[8. Annex I 26](#_Toc83643511)

[8.1. Virtual environment 26](#_Toc83643512)

[9. Annex II 27](#_Toc83643513)

[9.1. PyCharm 27](#_Toc83643514)

[10. Annex III 28](#_Toc83643515)

[10.1. SQLITE 28](#_Toc83643516)

[10.2. SQLite and Python types 28](#_Toc83643517)

[10.2.1. Introduction 28](#_Toc83643518)

[10.2.2. Using adapters to store additional Python types in SQLite databases 28](#_Toc83643519)

[10.2.3. Converting SQLite values to custom Python types 30](#_Toc83643520)

[10.2.4. Default adapters and converters 32](#_Toc83643521)

[11. Annex IV 34](#_Toc83643522)

[11.1. Logging 34](#_Toc83643523)

[12. Annex V 35](#_Toc83643524)

[12.1. Digital Ocean 35](#_Toc83643525)

# Global project organisation & structure

## Main blocks

The overall app can be split into 4 main parts:

1. Entry point (main)
2. Visual (dashboard)
3. Logic (controllers, managers, basic classes)
4. Binance/Simulator client

# Session & Session Manager

## Socket callbacks

### From Binance to ClientManager class

There are different websockets created to connect with Binance:

* One for the User socket
* One for each symbol session for the Symbol Ticker socket

The first passes the \_user\_socket\_callback() function and the other ones all have the same function callback \_symbol\_ticker\_socket\_callback().

Since this class is the interface with Binance and FakeClient class, these two methods only check the reference is not ***None***.

def \_user\_socket\_callback(self, msg: Dict):  
 # check it is a valid reference  
 if self.\_user\_callback:  
 self.\_user\_callback(msg)

def \_symbol\_ticker\_socket\_callback(self, msg: Dict):  
 # check it is a valid reference  
 if self.\_symbol\_ticker\_callback:  
 self.\_symbol\_ticker\_callback(msg)

### From ClientManager class to MarketSocketsIn class

Callbacks:

* Symbol ticker callback
* User callback

### From MarketSocketsIn class to SessionManager class

Callbacks:

* Symbol ticker callback
* Order traded callback
* Account balance callback
* Update previous callback
* Order canceled callback
* Order created callback

## Session

### Criteria to allow new perfect trade creation

There are different criteria groups:

* Related with enough liquidity (base & asset)
* Related with whether in one, or both assets, after this pt creation the remaining liquidity will be close to exhaustion.
* Related with predicted movements of market price (cmp)

Enough liquidity

As an overall principle, to allow the creation of a new perfect trade it is necessary that both assets (base & quote) have enough liquidity.

#### Base and quote liquidity are checked independently.

#### Liquidity close to exhaustion

#### Predicted market price movements

# Client & Client Manager

# Dashboard

# Binance API

Check order types available for a symbol:

1. info\_d = client.get\_exchange\_info()

It returns a dictionary with the following interesting items:

1. info\_d = client.get\_exchange\_info()
3. for symbol in info\_d[‘symbols’]:
4. if symbol[‘symbol’] == ‘BTCEUR’:
5. print(symbol[‘orderTypes’]

For the ‘BTCEUR’ symbol, the following order types are allowed:

* LIMIT
* LIMIT\_MAKER
* MARKET
* STOP\_LOSS\_LIMIT
* TAKE\_PROFIT\_LIMIT

The signature for **STOP\_LOSS\_LIMIT** and **TAKE\_PROFIT\_LIMIT** is different than for LIMIT, because it must include the ***stopPrice*** parameter

## Exchange info

**get\_exchange\_info()**[**[source]**](https://python-binance.readthedocs.io/en/latest/_modules/binance/client.html#Client.get_exchange_info)

Return rate limits and list of symbols

|  |  |
| --- | --- |
| **Returns:** | list - List of product dictionaries |

{

"timezone": "UTC",

"serverTime": 1508631584636,

"rateLimits": [

{

"rateLimitType": "REQUESTS",

"interval": "MINUTE",

"limit": 1200

},

{

"rateLimitType": "ORDERS",

"interval": "SECOND",

"limit": 10

},

{

"rateLimitType": "ORDERS",

"interval": "DAY",

"limit": 100000

}

],

"exchangeFilters": [],

"symbols": [

{

"symbol": "ETHBTC",

"status": "TRADING",

"baseAsset": "ETH",

"baseAssetPrecision": 8,

"quoteAsset": "BTC",

"quotePrecision": 8,

"orderTypes": ["LIMIT", "MARKET"],

"icebergAllowed": false,

"filters": [

{

"filterType": "PRICE\_FILTER",

"minPrice": "0.00000100",

"maxPrice": "100000.00000000",

"tickSize": "0.00000100"

}, {

"filterType": "LOT\_SIZE",

"minQty": "0.00100000",

"maxQty": "100000.00000000",

"stepSize": "0.00100000"

}, {

"filterType": "MIN\_NOTIONAL",

"minNotional": "0.00100000"

}

]

}

]

}

|  |  |
| --- | --- |
| **Raises:** | BinanceRequestException, BinanceAPIException |

## Get symbol info

**get\_symbol\_info(*symbol*)**[**[source]**](https://python-binance.readthedocs.io/en/latest/_modules/binance/client.html#Client.get_symbol_info)

Return information about a symbol

|  |  |
| --- | --- |
| **Parameters:** | **symbol** (*str*) – required e.g BNBBTC |
| **Returns:** | Dict if found, None if not |

{

"symbol": "ETHBTC",

"status": "TRADING",

"baseAsset": "ETH",

"baseAssetPrecision": 8,

"quoteAsset": "BTC",

"quotePrecision": 8,

"orderTypes": ["LIMIT", "MARKET"],

"icebergAllowed": false,

"filters": [

{

"filterType": "PRICE\_FILTER",

"minPrice": "0.00000100",

"maxPrice": "100000.00000000",

"tickSize": "0.00000100"

}, {

"filterType": "LOT\_SIZE",

"minQty": "0.00100000",

"maxQty": "100000.00000000",

"stepSize": "0.00100000"

}, {

"filterType": "MIN\_NOTIONAL",

"minNotional": "0.00100000"

}

]

}

|  |  |
| --- | --- |
| **Raises:** | BinanceRequestException, BinanceAPIException |

## Place order

1. >>> from src.market import Market, MarketMode
2. >>> mode = MarketMode.BINANCE
3. >>> market = Market(None, mode)
4. >>> order = market.\_client.order\_limit\_buy(symbol='BTCEUR', quantity=0.001, price='50000.0', newClientOrderId=’OR000001af3c’)
5. >>> order
6. {'symbol': 'BTCEUR', 'orderId': 555472500, 'orderListId': -1, 'clientOrderId': 'OR000001af3c', 'transactTime': 1618521179140, 'price': '50000.00000000', 'origQty': '0.00100000', 'executedQty': '0.00000000', 'cummulativeQuoteQty': '0.00000000', 'status': 'NEW', 'timeInForce': 'GTC', 'type': 'LIMIT', 'side': 'BUY', 'fills': []}

To cancel the previous order we can use either the orderId or the clientOrderId:

1. >>> result = market.\_client.cancel\_order(symbol='BTCEUR', orderId=555472500)
2. >>> result
3. >>> {'symbol': 'BTCEUR', 'origClientOrderId': ' OR000001af3c', 'orderId': 555472500, 'orderListId': -1, 'clientOrderId': 'jDBLGYh2g3a5urkkpwNwou', 'price': '50000.00000000', 'origQty': '0.00100000', 'executedQty': '0.00000000', 'cummulativeQuoteQty': '0.00000000', 'status': 'CANCELED', 'timeInForce': 'GTC', 'type': 'LIMIT', 'side': 'BUY'}

Once canceled, the result message shows the original clientOrderId as origClientOrderId

## Cancel order

python-binance:

Graphical user interface, text, application, email

Description automatically generated

Binance API:

Graphical user interface, application

Description automatically generated

**Response:** (Binance API)

{

"symbol": "LTCBTC",

**"origClientOrderId": "myOrder1"**,

"orderId": 4,

"orderListId": -1, //Unless part of an OCO, the value will always be -1.

"clientOrderId": "cancelMyOrder1",

"price": "2.00000000",

"origQty": "1.00000000",

"executedQty": "0.00000000",

"cummulativeQuoteQty": "0.00000000",

"status": "CANCELED",

"timeInForce": "GTC",

"type": "LIMIT",

"side": "BUY"

}

## Order traded

Received through the user socket (Order update) with **[‘e’] = ‘executionReport’**.

Other interesting items:

1. [‘x’] Execution type:
   1. NEW
   2. CANCELED
   3. REJECTED
   4. REPLACED
   5. TRADE
   6. EXPIRED
2. [‘X’] Order status:

Table

Description automatically generated with medium confidence

## Websockets

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generated

Steps:

1. Create a listen key
2. Ping/keep alive the listen key
3. Close the listen key
4. def \_start\_sockets(self):
5. # init socket manager
6. self.\_bsm = BinanceSocketManager(self.\_client)
8. # symbol ticker socket and get the **listen key**
9. self.\_symbol\_ticker\_s = self.\_bsm.start\_symbol\_ticker\_socket(
10. 'BTCEUR',
11. self.\_symbol\_ticker\_socket\_callback)
13. # user socket
14. self.\_user\_s = self.\_bsm.start\_user\_socket(
15. self.\_user\_socket\_callback)
17. # start sockets
18. self.\_bsm.start()
20. self.\_bsm.stop\_socket(self.\_symbol\_ticker\_s)
21. self.\_bsm.stop\_socket(self.\_user\_s)
23. # properly close the WebSocket, only if it is running
24. # trying to stop it when it is not running, will raise an error
25. if reactor.running:
26. reactor.stop()

## User socket

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

After placing an order, through ‘outboundAccountPosition’ we receive the updated balance of our cryptos:

* BTC
* BNB
* EUR

Each one would have to be updated in the dashboard in two ways:

* total
* for each active perfect trade

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

Text

Description automatically generated with low confidence

Graphical user interface, text, application, Teams

Description automatically generated

Text

Description automatically generated with medium confidence

## API errors

## Messages for -1010 ERROR\_MSG\_RECEIVED, -2010 NEW\_ORDER\_REJECTED, and -2011 CANCEL\_REJECTED

This code is sent when an error has been returned by the matching engine. The following messages which will indicate the specific error:

| **Error message** | **Description** |
| --- | --- |
| "Unknown order sent." | The order (by either orderId, clOrdId, origClOrdId) could not be found |
| "Duplicate order sent." | The clOrdId is already in use |
| "Market is closed." | The symbol is not trading |
| "Account has insufficient balance for requested action." | Not enough funds to complete the action |
| "Market orders are not supported for this symbol." | MARKET is not enabled on the symbol |
| "Iceberg orders are not supported for this symbol." | icebergQty is not enabled on the symbol |
| "Stop loss orders are not supported for this symbol." | STOP\_LOSS is not enabled on the symbol |
| "Stop loss limit orders are not supported for this symbol." | STOP\_LOSS\_LIMIT is not enabled on the symbol |
| "Take profit orders are not supported for this symbol." | TAKE\_PROFIT is not enabled on the symbol |
| "Take profit limit orders are not supported for this symbol." | TAKE\_PROFIT\_LIMIT is not enabled on the symbol |
| "Price \* QTY is zero or less." | price \* quantity is too low |
| "IcebergQty exceeds QTY." | icebergQty must be less than the order quantity |
| "This action disabled is on this account." | Contact customer support; some actions have been disabled on the account. |
| "Unsupported order combination" | The orderType, timeInForce, stopPrice, and/or icebergQty combination isn't allowed. |
| "Order would trigger immediately." | The order's stop price is not valid when compared to the last traded price. |
| "Cancel order is invalid. Check origClOrdId and orderId." | No origClOrdId or orderId was sent in. |
| "Order would immediately match and take." | LIMIT\_MAKER order type would immediately match and trade, and not be a pure maker order. |
| "The relationship of the prices for the orders is not correct." | The prices set in the OCO is breaking the Price rules. The rules are: SELL Orders: Limit Price > Last Price > Stop Price BUY Orders: Limit Price < Last Price < Stop Price |
| "OCO orders are not supported for this symbol" | OCO is not enabled on the symbol |
| "Quote order qty market orders are not support for this symbol." | MARKET orders using the parameter quoteOrderQty are not enabled on the symbol. |

## 

## -9xxx Filter failures

| **Error message** | **Description** |
| --- | --- |
| "Filter failure: PRICE\_FILTER" | price is too high, too low, and/or not following the tick size rule for the symbol. |
| "Filter failure: PERCENT\_PRICE" | price is X% too high or X% too low from the average weighted price over the last Y minutes. |
| "Filter failure: LOT\_SIZE" | quantity is too high, too low, and/or not following the step size rule for the symbol. |
| "Filter failure: MIN\_NOTIONAL" | price \* quantity is too low to be a valid order for the symbol. |
| "Filter failure: ICEBERG\_PARTS" | ICEBERG order would break into too many parts; icebergQty is too small. |
| "Filter failure: MARKET\_LOT\_SIZE" | MARKET order's quantity is too high, too low, and/or not following the step size rule for the symbol. |
| "Filter failure: MAX\_POSITION" | The account's position has reached the maximum defined limit. This is composed of the sum of the balance of the base asset, and the sum of the quantity of all open BUY orders. |
| "Filter failure: MAX\_NUM\_ORDERS" | Account has too many open orders on the symbol. |
| "Filter failure: MAX\_ALGO\_ORDERS" | Account has too many open stop loss and/or take profit orders on the symbol. |
| "Filter failure: MAX\_NUM\_ICEBERG\_ORDERS" | Account has too many open iceberg orders on the symbol. |
| "Filter failure: EXCHANGE\_MAX\_NUM\_ORDERS" | Account has too many open orders on the exchange. |
| "Filter failure: EXCHANGE\_MAX\_ALGO\_ORDERS" | Account has too many open stop loss and/or take profit orders on the exchange. |

# Python misc

## Command line arguments

1. pip install argparse
2. # Import the library  
   import argparse
3. # Create the parser  
   parser = argparse.ArgumentParser()
4. # Add an argument  
   parser.add\_argument('--name', type=str, required=True)
5. # Parse the argument  
   args = parser.parse\_args()
6. # Print "Hello" + the user input argument  
   print('Hello,', args.name)

## Exceptions

# Annex I

## Virtual environment

Create a new folder with the name of the project, create the virtual environment and activate it

1. > mkdir Polaris
2. > cd Polaris
3. > python3 -m venv .venv
4. > source .venv/bin/actívate
5. …
6. > deactivate

Copy into the new folder the file **requirements,txt** and install all packages:

1. *(.venv) crypto\_apps/dawning\_0.0.1*> cp requirements.txt ../Polaris
2. *(.venv) crypto\_apps/dawning\_0.0.1*> cd ../Polaris
3. *(.venv) crypto\_apps/Polaris*> pip install -r requirements.txt

To make a copy of the project in the new folder:

* From github download the .zip of the required project branch.
* Unzip the .zip into the new folder.
* Open atom and set this project as the project folder.
* Create new repository and publish to github.

Useful commands:

1. > pip freeze > requirements.txt
2. > pip list –outdated
3. > pip install –upgrade pip

Extra feature:

* In OS X Finder, **CMD + SHIFT + .** to toggle hidden files visibility

# Annex II

## PyCharm

Shortcuts:

* Open keymap: [CMD] + [,]

Setup log files visualization:

* Under Run>Edit configurations -> Logs, add the log file Polaris.log and an alias.
* This alias will appear in the running/debugging window when running from PyCharm (not if it is run from terminal).
* There is the option to select the log level live.

# Annex III

## SQLITE

## SQLite and Python types

### Introduction

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem:

| **Python type** | **SQLite type** |
| --- | --- |
| [None](https://docs.python.org/3/library/constants.html#None) | NULL |
| [int](https://docs.python.org/3/library/functions.html#int) | INTEGER |
| [float](https://docs.python.org/3/library/functions.html#float) | REAL |
| [str](https://docs.python.org/3/library/stdtypes.html#str) | TEXT |
| [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) | BLOB |

This is how SQLite types are converted to Python types by default:

| **SQLite type** | **Python type** |
| --- | --- |
| NULL | [None](https://docs.python.org/3/library/constants.html#None) |
| INTEGER | [int](https://docs.python.org/3/library/functions.html#int) |
| REAL | [float](https://docs.python.org/3/library/functions.html#float) |
| TEXT | depends on [text\_factory](https://docs.python.org/3/library/sqlite3.html" \l "sqlite3.Connection.text_factory" \o "sqlite3.Connection.text_factory), [str](https://docs.python.org/3/library/stdtypes.html#str) by default |
| BLOB | [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) |

The type system of the [sqlite3](https://docs.python.org/3/library/sqlite3.html#module-sqlite3) module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the [sqlite3](https://docs.python.org/3/library/sqlite3.html#module-sqlite3) module convert SQLite types to different Python types via converters.

### Using adapters to store additional Python types in SQLite databases

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must **adapt** them to one of the sqlite3 module’s supported types for SQLite: one of NoneType, int, float, str, bytes.

There are two ways to enable the [sqlite3](https://docs.python.org/3/library/sqlite3.html#module-sqlite3) module to adapt a custom Python type to one of the supported ones.

#### Letting your object adapt itself

This is a good approach if you write the class yourself. Let’s suppose you have a class like this:

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

Now you want to store the point in a single SQLite column. First you’ll have to choose one of the supported types to be used for representing the point. Let’s just use str and separate the coordinates using a semicolon. Then you need to give your class a method \_\_conform\_\_(self, protocol) which must return the converted value. The parameter protocol will be PrepareProtocol.

**import** **sqlite3**

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

**def** \_\_conform\_\_(self, protocol):

**if** protocol **is** sqlite3.PrepareProtocol:

**return** "*%f*;*%f*" % (self.x, self.y)

con = sqlite3.connect(":memory:")

cur = con.cursor()

p = Point(4.0, -3.2)

cur.execute("select ?", (p,))

print(cur.fetchone()[0])

con.close()

#### Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with [register\_adapter()](https://docs.python.org/3/library/sqlite3.html" \l "sqlite3.register_adapter" \o "sqlite3.register_adapter).

**import** **sqlite3**

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

**def** adapt\_point(point):

**return** "*%f*;*%f*" % (point.x, point.y)

sqlite3.register\_adapter(Point, adapt\_point)

con = sqlite3.connect(":memory:")

cur = con.cursor()

p = Point(4.0, -3.2)

cur.execute("select ?", (p,))

print(cur.fetchone()[0])

con.close()

The [sqlite3](https://docs.python.org/3/library/sqlite3.html#module-sqlite3) module has two default adapters for Python’s built-in [datetime.date](https://docs.python.org/3/library/datetime.html" \l "datetime.date" \o "datetime.date) and [datetime.datetime](https://docs.python.org/3/library/datetime.html" \l "datetime.datetime" \o "datetime.datetime) types. Now let’s suppose we want to store [datetime.datetime](https://docs.python.org/3/library/datetime.html" \l "datetime.datetime" \o "datetime.datetime) objects not in ISO representation, but as a Unix timestamp.

**import** **sqlite3**

**import** **datetime**

**import** **time**

**def** adapt\_datetime(ts):

**return** time.mktime(ts.timetuple())

sqlite3.register\_adapter(datetime.datetime, adapt\_datetime)

con = sqlite3.connect(":memory:")

cur = con.cursor()

now = datetime.datetime.now()

cur.execute("select ?", (now,))

print(cur.fetchone()[0])

con.close()

### Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let’s go back to the Point class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we’ll define a converter function that accepts the string as a parameter and constructs a Point object from it.

**Note**

Converter functions **always** get called with a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object, no matter under which data type you sent the value to SQLite.

**def** convert\_point(s):

x, y = map(float, s.split(b";"))

**return** Point(x, y)

Now you need to make the [sqlite3](https://docs.python.org/3/library/sqlite3.html#module-sqlite3) module know that what you select from the database is actually a point. There are two ways of doing this:

* Implicitly via the declared type
* Explicitly via the column name

Both ways are described in section [Module functions and constants](https://docs.python.org/3/library/sqlite3.html#sqlite3-module-contents), in the entries for the constants [PARSE\_DECLTYPES](https://docs.python.org/3/library/sqlite3.html#sqlite3.PARSE_DECLTYPES) and [PARSE\_COLNAMES](https://docs.python.org/3/library/sqlite3.html#sqlite3.PARSE_COLNAMES).

The following example illustrates both approaches.

**import** **sqlite3**

**class** **Point**:

**def** \_\_init\_\_(self, x, y):

self.x, self.y = x, y

**def** \_\_repr\_\_(self):

**return** "(*%f*;*%f*)" % (self.x, self.y)

**def** adapt\_point(point):

**return** ("*%f*;*%f*" % (point.x, point.y)).encode('ascii')

**def** convert\_point(s):

x, y = list(map(float, s.split(b";")))

**return** Point(x, y)

*# Register the adapter*

sqlite3.register\_adapter(Point, adapt\_point)

*# Register the converter*

sqlite3.register\_converter("point", convert\_point)

p = Point(4.0, -3.2)

*#########################*

*# 1) Using declared types*

con = sqlite3.connect(":memory:", detect\_types=sqlite3.PARSE\_DECLTYPES)

cur = con.cursor()

cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))

cur.execute("select p from test")

print("with declared types:", cur.fetchone()[0])

cur.close()

con.close()

*#######################*

*# 1) Using column names*

con = sqlite3.connect(":memory:", detect\_types=sqlite3.PARSE\_COLNAMES)

cur = con.cursor()

cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))

cur.execute('select p as "p [point]" from test')

print("with column names:", cur.fetchone()[0])

cur.close()

con.close()

### Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name “date” for [datetime.date](https://docs.python.org/3/library/datetime.html" \l "datetime.date" \o "datetime.date) and under the name “timestamp” for [datetime.datetime](https://docs.python.org/3/library/datetime.html" \l "datetime.datetime" \o "datetime.datetime).

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

**import** **sqlite3**

**import** **datetime**

con = sqlite3.connect(":memory:", detect\_types=sqlite3.PARSE\_DECLTYPES|sqlite3.PARSE\_COLNAMES)

cur = con.cursor()

cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()

now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))

cur.execute("select d, ts from test")

row = cur.fetchone()

print(today, "=>", row[0], type(row[0]))

print(now, "=>", row[1], type(row[1]))

cur.execute('select current\_date as "d [date]", current\_timestamp as "ts [timestamp]"')

row = cur.fetchone()

print("current\_date", row[0], type(row[0]))

print("current\_timestamp", row[1], type(row[1]))

con.close()

If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.

# Annex IV

## Logging

A picture containing text, newspaper, document

Description automatically generated

# Annex V

## Digital Ocean

Account:

Log in with google

* Ip: 167.71.62.77
* Usr: root
* Pwd: digi\*\*\*\*

1. \*\*\*\*\*\*\*\*\*\* Connect to Digital Ocean as root \*\*\*\*\*\*\*\*\*\*
2. $ ssh root@167.71.62.77
3. $ password:
5. root@droplet-002:~#
6. apt-get update
7. apt-get upgrade
8. \*\*\*\*\*\*\*\*\*\* Create user (first time) \*\*\*\*\*\*\*\*\*\*
9. # create user (pwd: pola\*\*\*\*)
10. adduser xavi
11. # add user to sudo
12. adduser xavi sudo
13. Exit
14. \*\*\*\*\*\*\*\*\*\* Connect to Digital Ocean as xavi \*\*\*\*\*\*\*\*\*\*
15. # connect with new user
16. ssh [xavi@167.71.62.77](mailto:xavi@167.71.62.77)
17. \*\*\*\*\*\*\*\*\*\* First time setup \*\*\*\*\*\*\*\*\*\*\*
18. # check Python versión
19. python3 –version
20. Python 3.8.5
21. # update to 3.9
22. sudo apt install python3.9
23. # install pip
24. sudo apt install pyton3-pip
25. # create Project directory
26. mkdir polaris\_plus\_project
28. # create virtual environment
29. cd polaris\_plus\_project
30. python3 -m venv .venv
31. # activate venv
32. source .venv/bin/actívate
33. # upload Project fron github
34. (.venv) git clone <https://github.com/xavibenavent/polaris_plus.git>
35. ls
36. cd polaris\_plus
37. # install packages needed
38. (.venv) pip install -r requirements.txt
39. # create required directories not in git
40. cd src
41. mkdir log
42. mkdir database
43. \*\*\*\*\*\*\*\*\*\* Run the app (test) \*\*\*\*\*\*\*\*\*\*
44. # run the app from the working directory
45. (.venv) python src/main.py --client\_mode=simulated --new\_master\_session=True

Setup the gunicorn server (WGSI), that will substitute the flask server

1. pip install gunicorn
2. # modify the app
3. # 1.run from cli and launch the dash module: pp\_dash.py
4. # 2.after app = dash.Dash(\_\_name\_\_, …) set the variable used in the cli
5. # server = app.server
6. # 3. Run
7. gunicorn src.pp\_dash:server

Setup the web server (nginx)

1. # install nginx
2. sudo apt install nginx

“Let's start by installing it. From the command line, while logged in to your server, run the following command:

sudo apt install nginx

We now want to create a configuration file for our app. Installing nginx does several things, one of which is creating a sites-enabled folder. We want to create our configuration file there, with basic options. We can use any text editor for that; a simple one to use that you can usually find on Linux machines is the nano editor. Running it as a command followed by a filename opens that file for editing (or creates one if it doesn't exist).

From the command line, run the following command to open and edit our file:

sudo nano /etc/nginx/sites-enabled/dash\_app

You should get an empty file, and you can copy and paste the following code, but make sure to replace the IP address after server\_name with your own IP address:

server {

    listen 80;

    server\_name 172.105.72.121;

    location / {

        proxy\_pass http://127.0.0.1:8000;

        proxy\_set\_header Host $host;

        proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

    }

  }

”

Excerpt From: Elias Dabbas. “Interactive Dashboards and Data Apps with  Plotly and Dash”. Apple Books.

“This code contains configuration for the server context, as you can see. It tells it to listen on port 80, which is the default port for web servers. It also defines the server\_name as the IP address. Later on, you can use this to define your own domain name.

It then defines the behavior of the server for location / under another block. The most important thing for us is that we are making nginx a proxy server with the proxy\_pass directive and telling it to listen to the URL and port that Gunicorn is listening to. So, now, the cycle should be complete. Our web server will be sending and receiving requests and responses through the correct URL and port, where the interface with our Python code is handled by Gunicorn.

Installing nginx creates a default configuration file, which we need to unlink with the following command:

sudo unlink /etc/nginx/sites-enabled/default

We just need to reload nginx after making this change. Keep this in mind when you make any changes in the future. You should reload nginx for any changes to take place, which you can do with the following command:

sudo nginx -s reload

Now[…]”

Excerpt From: Elias Dabbas. “Interactive Dashboards and Data Apps with  Plotly and Dash”. Apple Books.