

Projektbericht  
Studiengang : Informatik

---

# **Algorithmisches Handeln von Kryptowährungen**

von

Sebastian Flum

76855

Betreuender Mitarbeiter : Sebastian Stigler

Einreichungsdatum : 28. Februar 2021

# Eidesstattliche Erklärung

Hiermit erkläre ich, **Sebastian Flum**, dass ich die vorliegenden Angaben in dieser Arbeit wahrheitsgetreu und selbständig verfasst habe.

Weiterhin versichere ich, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, dass alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Westhausen, 28.02.2021

Ort, Datum

A handwritten signature in black ink, appearing to read 'S. Flum', with a large, stylized loop at the end.

Unterschrift (Student)

# Kurzfassung

Beim algorithmischen Handeln werden Handelsentscheidungen auf der Grundlage zuvor definierter Anweisungen und Regeln getroffen, die in Form eines Computerprogramms umgesetzt wurden. Dabei wird Code geschrieben, der die Trades im Namen des Händlers oder Investors ausführt, wenn bestimmte Bedingungen erfüllt sind.

Im Rahmen dieser Arbeit wurde versucht, jenes Verfahren auf den Handel von kryptographischen Währungen wie Bitcoin oder Ethereum anzuwenden und den Grundstein eines algorithmischen Handelssystems für diese zu entwerfen. So soll herausgefunden werden, ob der automatisierte Handel von Kryptowährungen, durch ein selbst entwickeltes System, erfolgreich sein kann.

Diese Arbeit beschäftigt sich im grundlegenden mit den folgenden Konzepten:

1. Algorithmische Handelsstrategien und technische Indikatoren
2. Implementierung einer Schnittstelle für Handelsplattformen
3. Backtesting von Handelsstrategien
4. Entwurf und Verwaltung eines Trading Bots
5. Erstellung einer Konsolenanwendung
6. Verknüpfung dieser Elemente innerhalb eines modularen Systems

Umgesetzt wurde dieses Projekt fast ausschließlich unter der objektorientierten Verwendung der Programmiersprache Python, in Verbindung mit einigen wenigen HTML Elementen.

# Inhaltsverzeichnis

<b>Eidesstattliche Erklärung</b>	<b>i</b>
<b>Kurzfassung</b>	<b>ii</b>
<b>Inhaltsverzeichnis</b>	<b>iii</b>
<b>Abbildungsverzeichnis</b>	<b>vii</b>
<b>Quelltextverzeichnis</b>	<b>ix</b>
<b>Abkürzungsverzeichnis</b>	<b>x</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Problemstellung und -abgrenzung . . . . .	2
1.3. Ziel der Arbeit . . . . .	3
<b>2. Grundlagen</b>	<b>4</b>
2.1. Trading . . . . .	4
2.1.1. Order Types . . . . .	4
2.1.2. Algorithmischer Handel . . . . .	5
2.1.3. Candlestick Charts . . . . .	5
2.1.4. Technische Indikatoren . . . . .	7
2.1.5. Trading Strategien . . . . .	8

2.2. Kryptowährungen . . . . .	9
2.2.1. Die Blockchain . . . . .	9
2.2.2. Mining . . . . .	10
2.2.3. Handel von Kryptowährungen . . . . .	11
2.3. Application Programming Interface . . . . .	12
2.3.1. REST API . . . . .	12
2.4. Python . . . . .	13
2.4.1. Pandas DataFrame . . . . .	14
2.5. HTML . . . . .	15
<b>3. Problemanalyse</b>	<b>16</b>
3.1. Konzeption einer Schnittstelle für Handelsplattformen . . . . .	16
3.1.1. Auswahl einer Handelsplattform . . . . .	16
3.1.2. Schnittstelle der API . . . . .	18
3.2. Algorithmische Handelsstrategien und technische Indikatoren . . . . .	18
3.3. Backtesting von Trading Strategien . . . . .	20
3.4. Entwurf und Verwaltung eines Trading Bots . . . . .	21
3.5. Erstellung einer Konsolenanwendung . . . . .	22
3.6. Zusammenfassung . . . . .	23
<b>4. Lösungskonzept</b>	<b>24</b>
4.1. Schnittstelle für Handelsplattformen . . . . .	24
4.1.1. Auswahl einer Handelsplattform . . . . .	24
4.1.2. Entwurf einer Schnittstelle . . . . .	24
4.2. Technische Indikatoren . . . . .	26
4.3. Algorithmische Trading Strategien . . . . .	27
4.3.1. Einbindung von Trading Strategien . . . . .	27

4.4. Backtest . . . . .	28
4.4.1. Entwurf eines Backtests . . . . .	28
4.4.2. Funktionsweise des Backtest . . . . .	29
4.4.3. Visualisieren des Backtests . . . . .	31
4.5. Entwurf und Verwaltung eines Trading Bot . . . . .	33
4.6. Entwurf einer Konsolenanwendung . . . . .	34
4.6.1. User Interface . . . . .	34
4.6.2. Zusammenspiel mit anderen Komponenten . . . . .	35
<b>5. Implementierung</b>	<b>37</b>
5.1. Binance Schnittstelle . . . . .	37
5.1.1. Vorgehen . . . . .	37
5.1.2. Klassendefinition . . . . .	37
5.1.3. API-Anfragen . . . . .	38
5.1.4. Sammeln von Candlestick-Daten . . . . .	40
5.2. Indikatoren . . . . .	43
5.2.1. Abstrakte Basisklasse Indicator . . . . .	43
5.2.2. Simple Moving Average . . . . .	44
5.3. Moving Average Strategie . . . . .	46
5.4. Backtest . . . . .	48
5.4.1. Ablauf des Backtests . . . . .	48
5.4.2. Erstellung eines Dashboards . . . . .	49
5.5. Trading Bot . . . . .	53
5.5.1. Repräsentation der Marktdaten . . . . .	54
5.5.2. Beschaffen und Aktualisieren der Marktdaten . . . . .	55
5.6. Bot Runner . . . . .	56

5.7. Konsolenanwendung . . . . .	56
5.7.1. Implementierung der Optionsauswahl . . . . .	57
5.7.2. Validierung der Nutzereingaben . . . . .	58
5.7.3. Erstellung von Backtests und Bots . . . . .	60
<b>6. Evaluierung</b>	<b>62</b>
<b>7. Zusammenfassung und Ausblick</b>	<b>63</b>
7.1. Erreichte Ergebnisse . . . . .	63
7.2. Ausblick . . . . .	63
<b>Literatur</b>	<b>65</b>
<b>A. Anhang zur Konsolenanwendung</b>	<b>68</b>
A.1. Main Menu . . . . .	68
A.2. Erstellung eines Backtests . . . . .	69
A.3. Erstellung eines Trading Bots . . . . .	74
A.4. Anzeigen der erstellten Trading Bots . . . . .	78

# Abbildungsverzeichnis

2.1. Kerzen eines Candlestick Charts . . . . .	6
2.2. Beispiel eines Candlestick Chart[2] . . . . .	7
2.3. Simple Moving Average (in Gelb) innerhalb eines Candlestick Charts .	8
2.4. Veranschaulichung einer Blockchain[34] . . . . .	10
2.5. Funktionsweise einer REST API[17] . . . . .	13
2.6. Veranschaulichung der grundlegenden Konzepte[25] . . . . .	14
2.7. Beispiel eines DataFrame[13] . . . . .	15
3.1. Veranschaulichung der grundlegenden Komponenten . . . . .	23
4.1. Konzept der REST API Schnittstelle . . . . .	25
4.2. Einfache Darstellung der Indikatoren durch ein Klassendiagramm . .	26
4.3. Einfache Darstellung von Strategien durch ein Klassendiagramm . . .	27
4.4. Vereinfachte Darstellung der Klasse <i>Backtest</i> innerhalb eines Klassen- diagramms . . . . .	29
4.5. Aktivitätsdiagramm für den Ablauf eines Backtest . . . . .	30
4.6. Erstellungsvorgang eines Dashboards . . . . .	32
4.7. Konzept eines Trading Bot und dessen Verwaltung als UML Diagramm	33
4.8. User Interface der Konsolenanwendung . . . . .	35
4.9. Die Konsolenanwendung dargestellt als Klassendiagramm . . . . .	36
5.1. Candlestick-Daten innerhalb eines DataFrame . . . . .	43
5.2. Berechnen und Hinzufügen des SMA Indikator an einen DataFrame	46



5.3. Ergebnis eines Ausschnitts von HTML Code . . . . .	51
5.4. Dashboard Ordnerstruktur . . . . .	53
5.5. Optionsauswahl des Main Menu . . . . .	58
5.6. Beispiel eines Validierungsfehlers bei fehlerhafter Eingabe . . . . .	60
A.1. Auswählen der API . . . . .	68
A.2. Auswählen der API . . . . .	69
A.3. Auswählen des Trading Symbols . . . . .	69
A.4. Auswählen der Strategie . . . . .	70
A.5. Auswählen der Dauer des Backtests . . . . .	70
A.6. Eingabe des Startkapitals . . . . .	71
A.7. Eingabe der Kaufmenge pro Kauf . . . . .	71
A.8. Ausgabe der Konfiguration . . . . .	72
A.9. Ausgabe der Backtest-Statistik . . . . .	73
A.10.Auswählen der API . . . . .	74
A.11.Auswählen der API . . . . .	74
A.12.Auswählen des Trading Symbols . . . . .	75
A.13.Auswählen der Strategie . . . . .	75
A.14.Eingabe des Startkapitals . . . . .	76
A.15.Eingabe der Kaufmenge pro Kauf . . . . .	76
A.16.Optimale Eingabe einer Beschreibung . . . . .	77
A.17.Überprüfen der Konfiguration . . . . .	77
A.18.Übersicht der Erstellten Trading Bots . . . . .	78

# Listings

2.1. Beispiel eines HTML Dokuments . . . . .	15
5.1. Binance Klassendefinition . . . . .	37
5.2. Binance API Calls . . . . .	38
5.3. Binance API Calls Fortsetzung . . . . .	39
5.4. Sammeln von Candlestick-Daten . . . . .	40
5.5. Candlestick-Daten in Rohform . . . . .	41
5.6. Sammeln von Candlestick-Daten Fortsetzung . . . . .	42
5.7. Implementierung der abstrakten Basisklasse <i>Indicator</i> . . . . .	43
5.8. Klassendefinition SimpleMovingAverage . . . . .	44
5.9. Implementierung der Methode <i>add_data</i> . . . . .	45
5.10. Klassendefinition der Moving Average Strategy . . . . .	47
5.11. Überprüfen der Kaufbedingung . . . . .	47
5.12. Vereinfachte Darstellung der Methode <i>run()</i> . . . . .	48
5.13. Erzeugung eines Candlestick Chart . . . . .	50
5.14. HTML Vorlage . . . . .	51
5.15. Substitution von HTML Variablen . . . . .	52
5.16. Erzeugen einer Ordnerstruktur für Backtest Dashboards . . . . .	52
5.17. Klassendefinition des Trading Bots . . . . .	53
5.18. Repräsentation der Marktdaten . . . . .	54
5.19. Beschaffung der Marktdaten . . . . .	55
5.20. Klassendefinition des Bot Runners . . . . .	56
5.21. Definition der Header . . . . .	57
5.22. Implementierung der Optionsauswahl . . . . .	57
5.23. Implementierung zur Validierung von Nutzereingaben . . . . .	59
5.24. Erstellung eines neuen Backtest . . . . .	60

# Abkürzungsverzeichnis

<b>API</b> Application Programming Interface . . . . .	2
<b>SMA</b> Simple Moving Average . . . . .	7
<b>CFD</b> Contract for Difference . . . . .	12
<b>REST</b> Representational State Transfer . . . . .	12
<b>HTTP</b> Hypertext Transfer Protocol . . . . .	13
<b>Pandas</b> Python Data Analysis Library . . . . .	14
<b>URL</b> Uniform Resource Locator . . . . .	13
<b>JSON</b> JavaScript Object Notation . . . . .	40
<b>ABC</b> Abstract Base Class . . . . .	43
<b>HTML</b> Hypertext Markup Language . . . . .	15

# 1. Einleitung

"Neues Allzeithoch: Bitcoin-Rekordjagd geht ungebremst weiter"[9]. "Kryptowährung: Bitcoin knackt Marke von 50.000 Dollar"[23]. Artikel wie diese sind schon lange keine Seltenheit mehr. Kryptowährungen, allen voran der Bitcoin, entwickelten sich in den letzten Jahren zum Sinnbild schnellen Reichtums. Doch das war nicht immer so. Im Jahre 2011, als sich der Bitcoin noch in seinen Kinderschuhen befand, kostete ein Coin gerade einmal 1 Dollar[5].

Vielleicht erwischt man sich nun selbst bei dem Gedanken daran, was wohl wäre, wenn man zu dieser Zeit schon das Potential dieser, damals unscheinbaren, Internetwährung erkannt hätte. Einige vorausdenkende Köpfe erkannten dies frühzeitig und wurden dadurch reich. Andere wiederum nutzten die neu entdeckte Kryptowährung dazu, online für ihre Pizza zu bezahlen. Verrückt, wenn man sich vor Augen führt, dass man mit der selben Menge Bitcoins heute zehntausende Pizzen bezahlen könnte. Vielleicht lässt dieser Gedanke die Herzen einiger Pizzaliebhaber höher schlagen, vorrangig stellt sich jedoch die Frage, ob man mit dem Besitz und Handel von Kryptowährungen heutzutage noch Erfolg haben kann. Und wenn ja - kann das auch ein Computerprogramm?

## 1.1. Motivation

Natürlich stellt die Aussicht auf realisierbaren Gewinn, erzeugt durch ein selbst entwickeltes Handelssystem, einen großen Motivationsfaktor dar - doch es ist mehr als das. Kryptowährungen liegen einem faszinierendem Stück Technologie zugrunde - der sogenannten **Blockchain** (dazu später mehr). Digitale Währungen können, wie jede herkömmliche Währung, getauscht und gehandelt werden, befinden sich gleichzeitig aber außerhalb der Kontrolle finanzieller Institutionen. Sie verbinden also die Welt des Finanzhandels mit der Welt der Informatik. Sie bietet die Möglichkeit eines interessanten Einblicks in beide Welten.

Abgesehen davon finde ich großen Gefallen daran, Lösungen für gegebene Problemstellungen zu entwerfen und in Programmcode umzusetzen. Es macht mir Spaß, neue Technologien kennenzulernen und deren Konzepte praktisch einzusetzen. Der Entwurf eines algorithmischen Handelssystems bietet dafür die perfekte Grundlage, da es beliebig erweitert werden kann und viele Schlüsselkonzepte vereint. So

stellt das Projekt eine interessante Möglichkeit für mich dar, Dinge wie objektorientierte Programmierung, Entwurf von Softwarearchitekturen und Datenbanken, Web-Entwicklung und vielem mehr, innerhalb eines zusammenhängenden Projektes, kennenzulernen und anzuwenden.

## 1.2. Problemstellung und -abgrenzung

Als Ganzes betrachtet ist die Problemstellung, ein algorithmisches Handelssystem zu entwerfen, relativ unüberschaubar. Zur Bewältigung muss es daher in mehrere Teilprobleme heruntergebrochen werden. Daraus ergeben sich folgende, im nachfolgenden grob beschriebene, Aufgabenstellungen:

### 1. Algorithmische Handelsstrategien und technische Indikatoren

Ohne eine geeignete Trading Strategie, die entscheidet, wann Einkäufe und Verkäufe zu tätigen sind, kann wohl kein System Erfolg haben. Diesen Trading Strategien liegen technische Indikatoren zu Grunde. Ein technischer Indikator dient dabei zur alternativen Darstellung der Kursverläufe und liefert der Strategie wertvolle Analyseinformationen. Der Fokus soll dabei jedoch weniger auf der Findung und Konzeption neuer Strategien und Indikatoren liegen, sondern darauf, wie diese möglichst einfach und modular in das System eingebunden werden können.

### 2. Entwurf einer Schnittstelle für Handelsplattformen

Der Handel von Kryptowährungen mittels eines algorithmischen Systems erfordert, genauso wie beim Handeln von Hand, eine Plattform, auf der die Ein- und Verkäufe getätigt werden. Im Vergleich zum Handeln von Hand, benötigt das Trading-System jedoch ein Application Programming Interface (API), an die es anknüpfen kann, um mögliche Aktionen der Plattform tätigen zu können. Einige Plattformen bieten solche APIs zum Teil kostenlos an. Die Problemstellung ergibt sich daraus, die von den Plattformen bereitgestellten APIs zu nutzen und eigene, möglichst modulare, Schnittstellen zur Verwendung dieser zu entwerfen.

### 3. Backtesting von Handelsstrategien

Backtesting bzw. Rückvergleich bezeichnet den Prozess, eine Strategie zu evaluieren, indem die Strategie auf historische Daten angewandt wird[28]. Findet man beispielsweise eine neue, vielversprechende Trading Strategie und möchte diese nun nach der Umsetzung in Code testen, ist es sehr riskant, die Strategie im Echtzeitbetrieb laufen zu lassen. Sollte die Strategie nämlich doch nicht so vielversprechend sein, läuft man Gefahr, große Verluste durch schlecht platzierte Ein- und Verkäufe hinnehmen zu müssen. Der Backtest wirkt diesem Risiko entgegen und stellt somit eine der wichtigsten Komponenten des Systems dar.

### 4. Entwurf und Verwaltung eines Trading Bots

Auswerten von Echtzeitdaten und automatisiertes Tätigen von Einkäufen und

Verkäufen. Da dies den Umfang dieser Arbeit jedoch übersteigt, soll innerhalb dieses Projekts nur der Grundstein dieses Features gelegt werden.

### **5. Erstellung einer Konsolenanwendung**

Alle Features des zu entwickelnden Systems, sollen dem Nutzer als Teil einer Konsolenanwendung zur Verfügung stehen. Mit dieser soll es beispielsweise möglich sein, Backtests sowie neue "Händler-Instanzen" aka Trading-Bots zu erstellen, welche verschiedenen Parametern wie Auswahl der Kryptowährung, Strategie, Kapital, usw. zu Grunde liegen.

### **6. Verknüpfung der Komponenten innerhalb eines modularen Systems**

Im Laufe der Arbeit werden Lösungskonzepte für die hier aufgeführten Problemstellungen entworfen und implementiert. Wichtig ist es dabei, die Komponenten mit Hinblick auf das Gesamtsystem und dessen Erweiterbarkeit/Modularität zu entwerfen, sowie ein sauberes und klar definiertes Zusammenspiel zu ermöglichen.

## **1.3. Ziel der Arbeit**

Das Ziel dieser Arbeit den Entwurf und die Implementierung eines algorithmischen Handelssystems für Kryptowährungen dar. Dabei soll es dem System möglich sein, Marktdaten über Kryptowährungen zu sammeln und automatisiert auszuwerten. Des weiteren soll das System über eine Teststrategie sowie den dazugehörigen technischen Indikatoren enthalten, welche durch einen, ebenfalls im System implementierten, Backtest ausgewertet werden kann. Das System soll die Möglichkeit haben, ausgewählte Marktdaten und Backtest-Ergebnisse zu visualisieren und anschaulich darzustellen. Unter anderem soll das System einen Prototyp eines Trading-Bots enthalten, welcher mit zukünftigen Erweiterungen in der Lage sein soll, echte Handelsaktionen durchzuführen. Diese Handelsaktionen werden innerhalb dieser Arbeit jedoch nicht implementiert, d.h. das System ist innerhalb dieses Projekts nicht in der Lage, eigenständig reale Käufe und Verkäufe zu tätigen. Die zuvor genannten Funktionen sollen dem Nutzer letztendlich durch eine Konsolenanwendung bereitgestellt werden.

## 2. Grundlagen

### 2.1. Trading

Das Wort "Trading", zu Deutsch: Handel, stammt aus dem Englischen und beschreibt den Kauf und Verkauf von Finanzprodukten an einer Börse. Menschen, die dieser Beschäftigung nachgehen, werden daher auch als Trader bezeichnet. Ein durchgeführter Handel heißt dementsprechend also Trade. (vgl. [24])

Trading bedeutet, die Schwankungen der Finanzmärkte mithilfe von Trades für die eigenen Zwecke zu nutzen. Das heißt Kaufen, Verkaufen und dabei Gewinne einstreichen - oder den Verlust verkraften. Durch das Internet und benutzerfreundliche Trading-Anbieter ist es auch für Hobby-Anleger möglich, mit Finanzinstrumente (z.B. Wertpapiere, Währungen, Rohstoff-Zertifikate o.ä.) zu handeln. Beim Trading handelt es sich grundsätzlich um Spekulation. Wo ein Trader sein Geld investiert, ist meist von untergeordneter Wichtigkeit. Beispielsweise geht es nicht darum, Anteile eines Unternehmens zu kaufen, um langfristig an dessen Entwicklung teilzuhaben. Das Ziel eines Traders ist es, durch einen Kursanstieg im Zeitraum des Besitzes eines Finanzinstruments Gewinn zu erzielen. Das bedeutet, ein Trader kauft beispielsweise eine Aktie, hofft auf einen Kursanstieg und verkauft sie dann wieder. Die anschließende Wertdifferenz abzüglich der Transaktionskosten stellt den Gewinn des Traders dar. (vgl. [8])

#### 2.1.1. Order Types

Mit der Verbreitung der digitalen Technologie und des Internets entscheiden sich viele Anleger dafür, Aktien selbst online zu kaufen und zu verkaufen, anstatt Beratern hohe Provisionen für die Ausführung von Geschäften zu zahlen. Beim Platzieren eines Auftrags bzw. Orders stehen dem Trader mehrere verschiedene Order-Types zur Verfügung.

##### **Market Order**

Eine Market Order ist die grundlegendste Art des Handels. Es ist ein Auftrag zum sofortigem Kauf oder Verkauf zum aktuellen Kurs. Kauft man eine Aktie, zahlt man in der Regel einen Preis zum (oder in der Nähe des veröffentlichten) Briefkurses. Wenn eine Aktie verkauft werden soll, erhält man einen Preis zum (oder in der Nähe

des eingestellten) Geldkurses.

### Stop-Loss Order

Eine Stop-Loss Order ist eine der nützlichsten Orders. Sie unterscheidet sich von anderen, denn im Gegensatz zur Market Order, die aktiv ist, sobald sie eingegeben werden, bleibt diese Order ruhend, bis ein bestimmter Preis überschritten wird. Erst dann wird sie als Market Order aktiviert.

Wenn zum Beispiel eine Stop-Loss Order auf die XYZ-Aktien bei 45€ pro Aktie platziert würde, wäre die Order inaktiv, bis der Preis 45€ erreicht oder unterschritten hat. Die Order würde dann in eine Market Order umgewandelt und die Aktien würden zum besten verfügbaren Preis verkauft werden. Man sollte diese Art von Order in Betracht ziehen, wenn man keine Zeit hat, den Markt ständig zu beobachten, aber Schutz vor einer großen Abwärtsbewegung benötigt. (vgl. [11])

## 2.1.2. Algorithmischer Handel

Algorithmischer Handel beschreibt den automatischen Handel von Finanzinstrumenten durch Computerprogramme. Dabei folgt das Computerprogramm einem vordefiniertem Satz von Anweisungen (Algorithmus), um einen Trade zu platzieren. Die Trades können theoretisch Gewinne mit einer Geschwindigkeit und Häufigkeit generieren, die für einen menschlichen Trader unmöglich sind. (vgl. [22])

Die definierten Sätze von Anweisungen basieren auf Timing, Preis, Menge und/oder einem beliebigen mathematischen Modell. Abgesehen von den Gewinnmöglichkeiten für den Trader, macht der algorithmische Handel die Märkte liquider und den Handel systematischer, da der Einfluss menschlicher Emotionen auf die Handelsaktivität ausgeschlossen wird. (vgl. [22])

## 2.1.3. Candlestick Charts

Ein Candlestick Chart ist ein Finanzdiagramm, mit dem sich die Kursbewegungen eines Wertpapiers oder Ähnlichem darstellen lassen. Es besteht aus einer Aneinanderreihung von sogenannten **Candles** (Kerzen), welche für die verschiedensten Zeiteinheiten verwendet werden können und vier zentrale Informationswerte enthalten:

- Eröffnungskurs (Open)
- Schlusskurs (Close)
- Höchstkurs (High)



- Tiefstkurs (Low)

(vgl. [1])

Die Candles haben gegenüber den früher gängigeren Balkencharts den Vorteil, optisch anschaulicher zu sein, ohne dabei gleich kompliziert zu werden. Grundsätzlich wird dabei zwischen zwei verschiedenen Kerzen unterschieden, welche in der nachfolgenden Abbildung (Abb. 2.1) dargestellt werden.

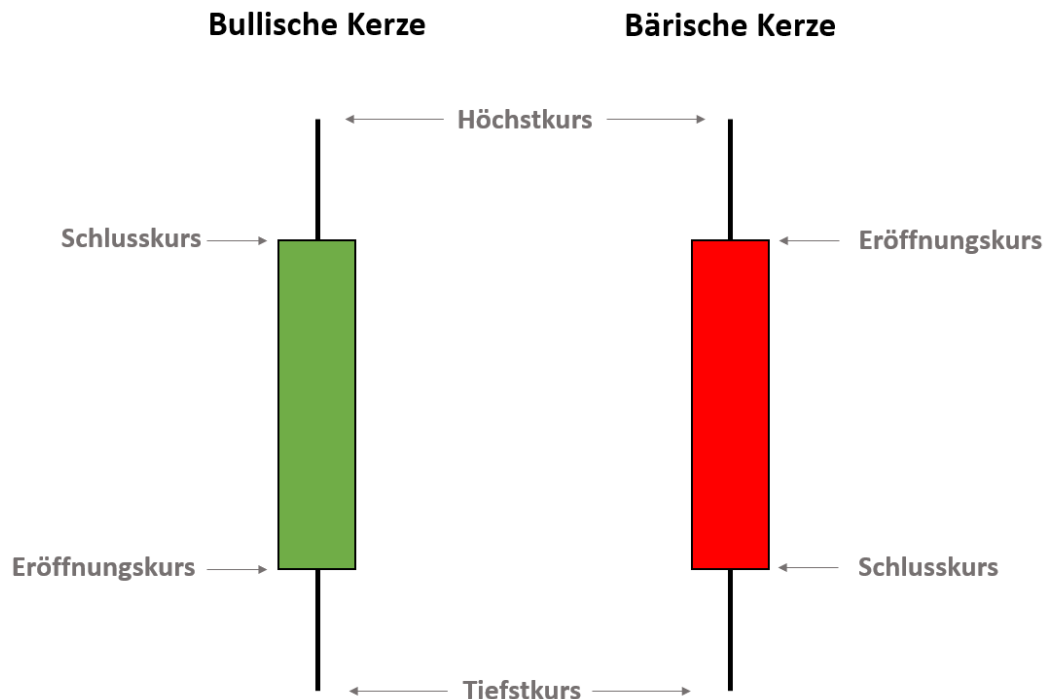


Abbildung 2.1.: Kerzen eines Candlestick Charts

Jede Kerze kann dabei für jede beliebige Zeitspanne stehen - für einen Tag, eine Woche, einen Monat, oder auch für kurzfristigere Zeitebenen auf Minutenbasis. Die Aussagen der Kerzen sind dabei jedoch immer dieselben.

Die Farbe der Candles lässt sofort erkennen, ob es sich um eine bullische, oder eine bärische Kerze handelt. Die **bullische Kerze** symbolisiert ein Zeitintervall, welches positiv verlief, weil der Schlusskurs über dem Eröffnungskurs des Intervalls lag. Analog dazu bedeutet die **bärische Kerze**, dass der Kursverlauf während des Zeitintervalls der Kerze negativ verlief, weil der Schlusskurs unter dem Eröffnungskurs lag. Wichtig ist es dabei, sich im Kopf zu behalten, dass jede Kerze immer nur eine Aussage über den Kursverlauf ihres Zeitintervalls darstellt. (vgl. [14])

Fügt man alle Candles zu einem Candlestick Chart zusammen erhält man ein Diagramm wie in Abbildung 2.2 dargestellt.



Abbildung 2.2.: Beispiel eines Candlestick Chart[2]

### 2.1.4. Technische Indikatoren

Technische Indikatoren sind Chart-Analyse-Tools, die Tradern helfen können, Preisbewegungen besser zu verstehen und darauf zu reagieren. Es gibt eine riesige Auswahl an technischen Analysewerkzeugen, die Trends analysieren, Preisdurchschnitte liefern, die Volatilität (Schwankung) messen und mehr. (vgl. [16])

Ein sehr beliebter technischer Indikator ist der **Moving Average**, zu Deutsch: gleitender Mittelwert, welcher benutzt wird, um den durchschnittlichen Schlusskurs des Marktes über einen bestimmten Zeitraum darzustellen. Trader machen oft Gebrauch von Moving Averages, da sie ein guter Hinweis auf die aktuelle Marktdynamik sein können. (vgl. [6])

#### Wie wird ein Moving Average berechnet?

Eine der gängigsten Moving Averages ist der einfache gleitende Mittelwert bzw. Simple Moving Average (SMA), welcher einfach den Durchschnitt aller Datenpunkte in der Serie geteilt durch die Anzahl der Punkte berechnet[6].

$$A = \{ a_1, a_2, \dots, a_n \}$$

$n$  = number of time periods / price elements

$$SMA = \frac{a_1 + a_2 + \dots + a_n}{n}$$

Betrachtet man zum Beispiel eine 5-Tage-SMA auf einem Tages-Chart von EUR/USD und die Schlusskurse über die letzten 5 Tage sind diese wie folgt.

Tag 1: 1.321

Tag 2: 1.301

Tag 3: 1.325

Tag 4: 1.327

Tag 5: 1.326

$$SMA = (1.321 + 1.301 + 1.325 + 1.327 + 1.326)/5 = 1.32$$

Veranschaulichen lässt sich dieser Indikator gut in einem Candlestick Chart, wie in Abbildung 2.3 dargestellt.



Abbildung 2.3.: Simple Moving Average (in Gelb) innerhalb eines Candlestick Charts

### 2.1.5. Trading Strategien

Eine Trading Strategie ist eine Methode zum Kaufen und Verkaufen auf Märkten, welche auf vordefinierten Regeln basiert, die zum Treffen von Handelsentscheidungen verwendet werden<sup>[4]</sup>.

Es gibt viele Arten von Handelsstrategien. Größtenteils basieren sie entweder auf technischen Daten oder auf Fundamentaldaten. Letzteres beschreibt langfristige,

grundlegende Informationen über die realen Produktionsmöglichkeiten, die Strukturen der Wirtschaft sowie den Vermögensstatus der Wirtschaftseinheiten, welche in dieser Arbeit jedoch nicht näher betrachtet werden[26]. Die Gemeinsamkeit ist, dass sich beide auf quantifizierbare Informationen stützen, die auf ihre Genauigkeit hin rückgetestet werden können. Technische Handelsstrategien verlassen sich auf technische Indikatoren, um Handelssignale zu generieren. Technische Trader glauben, dass alle Informationen über ein bestimmtes Wertpapier o.ä. in seinem Preis enthalten sind und dieser sich in Trends bewegt. (vgl. [4])

Eine einfache Trading Strategie kann zum Beispiel eine **Moving-Average-Strategy** sein, bei welcher ein Kaufsignal erzeugt wird, sobald der aktuelle Preis einen gewissen Wert unter dem der Moving Average liegt. Betrachtet man im Hinblick auf diese Strategie noch einmal Abbildung 2.3, lassen sich einige Stellen im Chart erkennen, bei denen diese Strategie wohl Kaufsignale erzeugen würde.

## 2.2. Kryptowährungen

Eine Kryptowährung ist eine digitale oder virtuelle Währung, die durch Kryptographie gesichert ist, was es nahezu unmöglich macht, sie zu fälschen oder doppelt auszugeben. Viele Kryptowährungen sind dezentralisierte Netzwerke, die auf der Blockchain-Technologie basieren - einem verteilten Hauptbuch, das von einem verteilten Netzwerk von Computern erzwingen wird. Ein entscheidendes Merkmal von Kryptowährungen ist, dass sie im Allgemeinen nicht von einer zentralen Behörde ausgegeben werden, was sie theoretisch immun gegen staatliche Eingriffe oder Manipulationen macht.

Die erste Blockchain-basierte Kryptowährung war Bitcoin, die immer noch die beliebteste und wertvollste ist. Heute gibt es tausende von alternativen Kryptowährungen mit verschiedenen Funktionen und Spezifikationen. Einige davon sind Klone oder Abspaltungen von Bitcoin, während andere neue Währungen sind, die von Grund auf neu entwickelt wurden. (vgl. [12])

### 2.2.1. Die Blockchain

Einfach ausgedrückt ist eine Blockchain nichts anderes als eine verteilte, öffentliche Datenbank[27], dessen Funktionsprinzip die meisten Kryptowährungen zugrunde liegen[29]. Um das Prinzip einer Blockchain verstehen zu können, hilft der Vergleich mit einem Gruppenchat:

Ben, Lisa und Mia haben sich um 14 Uhr im Park verabredet. Diese Information ist nach dem Absenden auf allen Smartphones im Chatverlauf gespeichert. Mia überlegt

es sich anders und möchte sich erst um 16 Uhr treffen. Diese Entscheidung kann sie aber nicht alleine fällen. Sie kann auch nicht einfach die Uhrzeit im Chatverlauf nachträglich ändern. Mia muss also entweder gemeinsam mit den anderen von vorne planen, oder sie steht um 16 Uhr alleine im Park. Niemand kann nachträglich etwas verändern. Dieses Prinzip ist der Kern einer Blockchain, denn auch hier werden Informationen in einem "Verlauf" gespeichert. Informationen sind hierbei hauptsächlich Transaktionsinformationen von Nutzern, die diese getätigt haben.

Jede Information bildet einen Block, für welche ein Hash - vergleichbar mit einem digitalen Fingerabdruck - berechnet wird. Zusätzlich enthält dieser Hash, den Hash des vorherigen Blocks. So werden die Blöcke über die Hashes, wie in Abbildung 2.4 veranschaulicht, zu einer Kette verbunden. Ähnlich wie bei einem Chatverlauf können die Informationen nicht verändert werden. Verändert sich nämlich die Information, verändert sich auch der Hash und die Kette würde auseinander brechen. Genau wie Mia und ihre Freunde mit der Planung von vorne beginnen müssten, müssten die Hashes aller folgenden Blöcke neu berechnet werden. (vgl. [34])

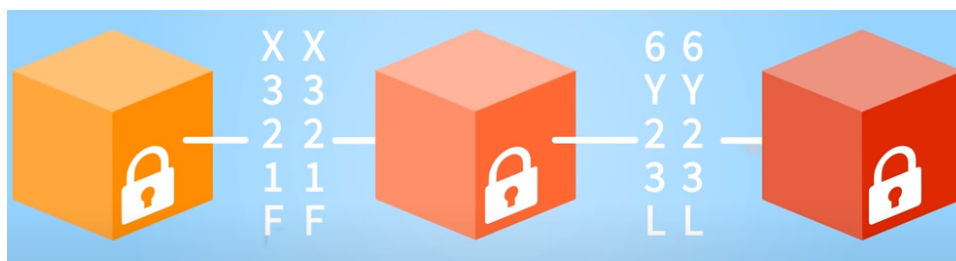


Abbildung 2.4.: Veranschaulichung einer Blockchain[34]

Die Blockchain wird über ein dezentrales Netzwerk verwaltet, dem jeder beitreten kann. Jedes Mitglied hat eine vollständige Kopie der kompletten Blockchain auf dem Computer, welche von diesem auch geprüft wird. Ein neuer Block wird erst hinzugefügt, wenn ihn jeder im Netzwerk verifiziert hat. Jeder kontrolliert sozusagen jeden, was die Blockchain damit sehr sicher macht. Wichtig ist dabei aber nicht der Mensch vor dem Computer, oder eine dritte Instanz wie beispielsweise ein Notar. Die Kontrolle, und somit das Vertrauen, werden von der Blockchain technisch hergestellt. (vgl. [34])

### 2.2.2. Mining

Wie das Geld herkömmlicher Währungen, wie beispielsweise Euro oder Dollar, entsteht, ist kein Geheimnis. Das Bargeld entsteht unter staatlicher Regie. Das Recht zur Prägung von Münzen liegt direkt beim Staat, während die staatliche Bundesbank Scheine herstellt. Beides müssen die privaten Geschäftsbanken bei der Bundesbank kaufen. (vgl. [15]). Doch wie funktioniert dies bei Kryptowährungen?

Instanzen einer Kryptowährung werden bei einem Prozess erzeugt, der **Mining**, zu Deutsch: Bergbau, genannt wird. Diese Analogie ist eigentlich recht passend, da neue Coins nur gefunden werden können, wenn ein bestimmter Arbeitsaufwand betrieben wird.

Der vorrangige Sinn des Minings ist jedoch nicht die Erschaffung von Coins, welcher eher als Nebeneffekt zu verstehen ist. Der Sinn des Mining ist vor allem das Verifizieren und Durchführen von Transaktionen sowie die Überprüfung, ob sich alle Teilnehmer des Netzwerks an dessen Regeln halten. Der Prozess des Minings wird dabei von Minern - also Computern mit Rechenleistung - durchgeführt. Wie bereits in Unterkapitel 2.2.1 beschrieben, enthalten die Blöcke gewisse Transaktionsinformationen. Zusätzlich zu diesen Transaktionsinformationen besitzt jeder Block eine **Coinbase Transaktion**, welche neue Coins enthält, die dann als "Belohnung" für das Sichern des Netzwerkes an die Miner ausgezahlt werden.

Ein Miner versucht im Endeffekt also, Transaktionen zu einem Block zu verbinden. Hat er diese Transaktionsinformationen zu einem passenden Block verbunden, wird er der Blockchain hinzugefügt und der Miner für seine Arbeitsleistung bezahlt. Veranschaulichen lässt sich dieser Prozess durch die Analogie eines Puzzles. Hierbei versucht der Miner, das Bild des Puzzles (Blockchain) zu bilden, indem er Puzzlestücke (Blöcke) zusammenfügt. Wer das passende Puzzlestück findet, erhält die Belohnung. (vgl. [3])

### 2.2.3. Handel von Kryptowährungen

Kryptowährungen werden in sogenannten **Wallets** gespeichert. Das sind digitale Geldbörsen, die auf den unterschiedlichsten digitalen Geräten wie beispielsweise einem Computer, Smartphone oder USB-Stick liegen können. Anders als beim klassischen Geldsystem, wo das Geld bei einer Bank und innerhalb eines Girokontos verwaltet wird, gehören diese Wallets nur dem Besitzer, was bedeutet, dass niemand diese Geldbörsen sperren kann. Auch viele andere Einschränkungen normaler Währungen kennt eine Kryptowährung nicht. So gibt es beispielsweise keine Überweisungslimits, maximale Transaktionshöhen oder geografische Einschränkungen. Überweisungen können problemlos von jedem Ort der Welt an jeden Ort der Welt gesendet werden, solange Sender und Empfänger Internetzugang haben und ein Wallet besitzen. (vgl. [10])

Wer sich nun dazu entschlossen hat mit Kryptowährungen zu handeln, hat generell zwei Möglichkeiten:

#### 1. Kryptowährungen kaufen

Der direkte Weg wäre Coins zu kaufen und diese solange zu halten, bis der erwünschte Kursanstieg erreicht wurde, um sie dann wieder zu verkaufen. Dazu wird jedoch, wie oben beschrieben, ein Wallet benötigt. Die Coins werden dann auf Exchange

Börsen oder Krypto-Marktplätzen im Internet gekauft und an das eigene Wallet geschickt.

## 2. Trading über CFDs

Alternativ besteht die Möglichkeit des Crypto Trading per Contract for Difference (CFD). Hierbei spekulieren Anleger lediglich auf den Anstieg bzw. den Fall des Kurses. Dabei können Basiswerte festgelegt werden, zu denen die CFDs gekauft bzw. verkauft werden sollen. Steigt die Kryptowährung im Wert um 10%, dann steigt auch der Wert der erworbenen CFDs um 10%. (vgl. [21])

## 2.3. Application Programming Interface

Ein Application Programming Interface, bzw. Programmierschnittstelle, ist ein Programmteil, der von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird. Die Programmanbindung findet dabei auf Quelltext-Ebene statt. (vgl. [32])

Beispiel:

Ein Restaurant bildet ein System. Kunden wollen dieses System nutzen, indem sie mithilfe einer Essenskarte Bestellungen aufgeben. Dabei ist die Küche der Teil des Systems, der die Bestellungen der Gäste zubereitet. Was fehlt, ist das entscheidende Bindeglied, um die Bestellung der Kunden an die Küche zu übermitteln und ihnen ihr Essen an den Tisch zu liefern. An dieser Stelle kommt der Kellner ins Spiel. Der Kellner ist der Bote, bzw. API, der die Anfrage der Kunden entgegennimmt und der Küche - bzw. dem System - mitteilt, was zu tun ist. Dann liefert der Kellner die Antwort an den Kunden zurück. In diesem Fall ist es das Essen. (vgl. [18])

### 2.3.1. REST API

REST steht für Representational State Transfer und beschreibt ein Paradigma (grundsätzliche Denkweise) für die Softwarearchitektur von verteilten Systemen, insbesondere für Webservices. Representational State Transfer (REST) ist eine Abstraktion der Struktur und des Verhaltens des World Wide Web und hat das Ziel, einen Architekturstil zu schaffen, der die Anforderungen des modernen Web besser darstellt. Der Zweck von REST liegt schwerpunktmäßig auf der Kommunikation zwischen Client und Server in Netzwerken. Eine REST API ist demnach also eine Programmierschnittstelle, die diesem Paradigma folgt. (vgl. [33])

Die Funktionsweise wird in Abbildung 2.5 ersichtlich.

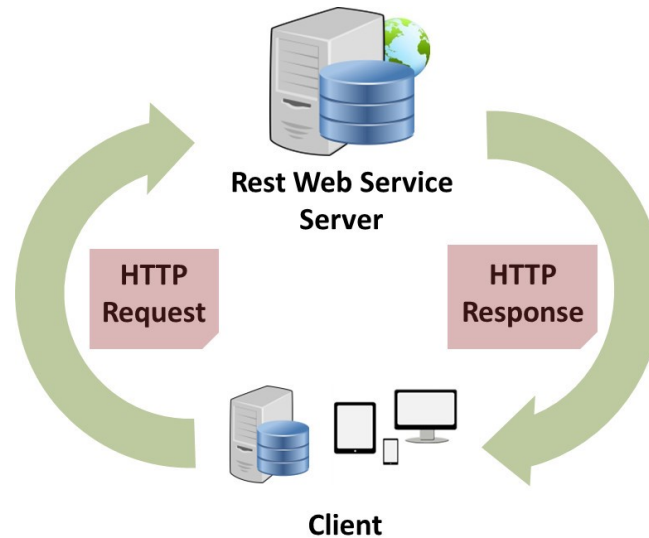


Abbildung 2.5.: Funktionsweise einer REST API[17]

Die API kommuniziert mit dem Server über das sogenannte Hypertext Transfer Protocol (HTTP) - ein Protokoll zur Übertragung von Daten über ein Rechnernetz. Es wird hauptsächlich eingesetzt, um Webseiten aus dem World Wide Web in einen Webbrowser zu laden[31]. Mittels HTTP Request, werden also Informationen vom Server angefordert, wodurch der Server wiederum eine HTTP Response erzeugt, die der Client dann erhält. Um eine Request an die API zu senden, werden Uniform Resource Locator (URL) benutzt - wie man sie aus der Verwendung eines Browsers kennt. Je nachdem, welche URL kontaktiert wird, führt die API unterschiedliche Aktionen aus, die unterschiedliche Antworten liefern.

## 2.4. Python

Technisch gesehen ist Python eine interpretierte, objektorientierte High-Level-Programmiersprache mit dynamischer Semantik. Ihre auf hoher Ebene eingebauten Datenstrukturen, kombiniert mit dynamischer Typisierung und dynamischer Bindung, machen sie sehr attraktiv für die schnelle Anwendungsentwicklung sowie für den Einsatz als Skript-Sprache, um bestehende Komponenten miteinander zu verbinden. Die einfach, leicht zu erlernende Syntax von Python betont die Lesbarkeit und reduziert somit die Kosten für die Programmwartung. Python unterstützt Module und Pakete, was die Modularität der Programme und die Wiederverwendung von Code fördert. Der Python-Interpreter und die umfangreiche Standardbibliothek sind in Quell- oder Binärform für alle wichtigen Plattformen kostenlos erhältlich und können frei verteilt werden. (vgl.[20])



Ein Python-Programm wird vom Python-Interpreter ausgeführt. Dieser stellt dabei eine umfangreiche Standardbibliothek bereit, die vom Programm verwendet werden kann. Außerdem erlaubt es die Python API einem externen C-Programm, den Interpreter zu verwenden oder zu erweitern. Dargestellt in Abbildung 2.6.

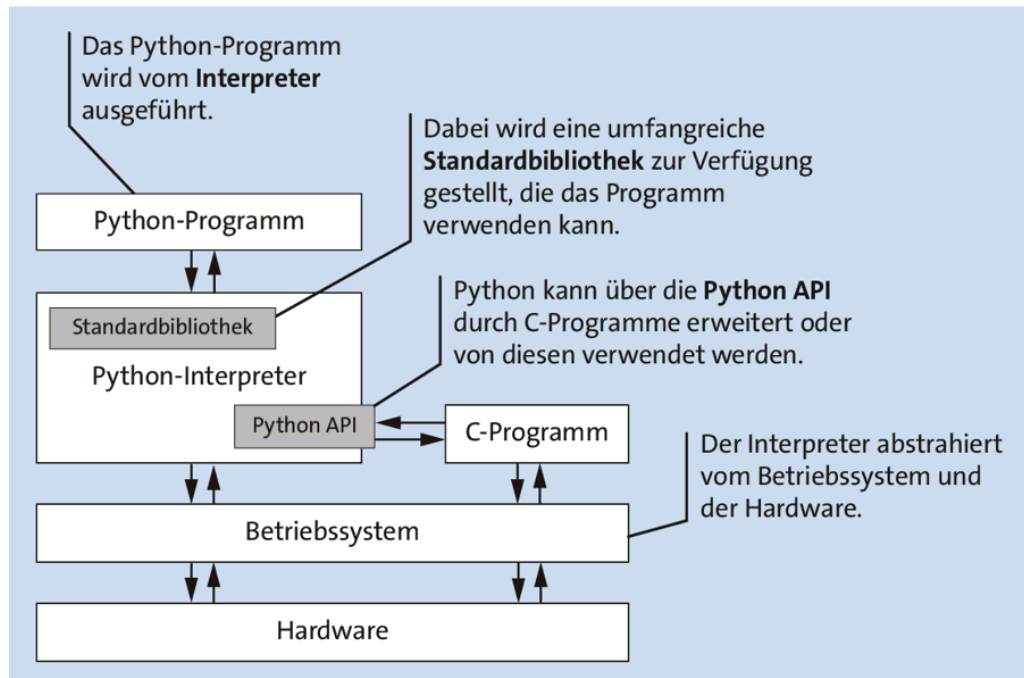


Abbildung 2.6.: Veranschaulichung der grundlegenden Konzepte[25]

### 2.4.1. Pandas DataFrame

Python Data Analysis Library (Pandas) ist ein beliebtes Python-Paket, das schnelle, flexible und ausdrucksstarke Datenstrukturen bereitstellt, um die Arbeit mit strukturierten (tabellarischen, mehrdimensionalen, potenziell heterogenen) Daten einfach und intuitiv zu gestalten. Es zielt darauf ab, ein grundlegender High-Level-Baustein für die praktische, reale Datenanalyse in Python zu sein. (vgl. [19])

Eine der wichtigsten, von pandas bereitgestellten, Datenstrukturen ist der **DataFrame**. Ein DataFrame ist eine zweidimensionale größenveränderliche, tabellarische Datenstruktur mit beschrifteten Achsen. Das bedeutet, die Daten sind tabellarisch in Zeilen und Spalten ausgerichtet, wie in Abbildung 2.7 dargestellt. (vgl. [13])

	<i>Name</i>	<i>Team</i>	<i>Number</i>	<i>Position</i>	<i>Age</i>
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Abbildung 2.7.: Beispiel eines DataFrame[13]

## 2.5. HTML

Die Hypertext Markup Language (HTML) ist eine textbasierte Auszeichnungssprache zur Strukturierung elektronischer Dokumente wie Texte mit Hyperlinks, Bildern und anderen Inhalten. Verwendet werden sie vor allem mithilfe von Web-Browsern, da sie die Grundlage der Inhalte des World Wide Web bilden. (vgl. [30])

Der Text wird durch sogenannte **Markups** strukturiert und die darin enthaltenen HTML-Elemente durch Tags markiert, d.h. durch einen Starttag und Endtag. Ein Starttag beginnt immer mit dem Zeichen < während er analog dazu mit dem Zeichen > geschlossen wird. Ein Endtag enthält zusätzlich einen Backslash. Damit lassen sich beispielsweise folgende Dokumentstrukturen erzeugen.

---

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Website Title</title>
5     <!-- more header information -->
6   </head>
7   <body>
8     <p>content of the website</p>
9   </body>
10  </html>

```

---

Quelltext 2.1: Beispiel eines HTML Dokuments

## 3. Problemanalyse

Wie bereits in Kapitel 1.2 grob beschrieben, befasst sich das Projekt und die Arbeit in erster Linie damit, ein algorithmisches Handelssystem zu entwerfen. Um dieses Problem angehen zu können, muss es vorab in mehrere Teilprobleme gebrochen werden, welche wiederum in noch kleinere Teilprobleme zerlegt werden. So wird klar, wie an das Problem herangegangen werden muss, welche Komponenten zu entwickeln sind und welche Abhängigkeiten zwischen diesen bestehen.

### 3.1. Konzeption einer Schnittstelle für Handelsplattformen

Wer Kryptowährungen handeln will, benötigt dazu eine Plattform, auf der Einkäufe und Verkäufe getätigt werden. Hat man sich dazu entschlossen am Handel von Kryptowährungen teilzunehmen, erstellt man sich dazu ein Konto bei einem der hunderten Crypto-Trading-Anbieter. Einmal getan, loggt man sich auf der Homepage oder der App des Anbieters ein und ist für den Handel von Kryptowährungen bereit. Im Vergleich zum Handeln von Hand, benötigt das Trading-System jedoch eine API, an die es anknüpfen kann, um mögliche Aktionen der Plattform tätigen zu können. Einige der Anbieter für Crypto-Trading stellen kostenlose APIs für die Nutzung mittels Code bereit.

#### 3.1.1. Auswahl einer Handelsplattform

Um sich für eine geeignete Plattform entscheiden zu können, bedarf es mehr als die Bereitstellung einer funktionstüchtigen API. Nach welchen Kriterien sollte eine Handelsplattform also ausgewählt werden?

Anforderungen an die Plattform:

##### 1. Gewährleistung der Sicherheit des Accounts

Die Sicherheit des Accounts und der damit verbundene Wert, der aus dem Besitz von Kryptowährungen hervorgeht hat höchste Priorität. Eine Bereitstellung von Zwei-Faktor-Authentifizierung sollte daher gegeben sein.

**2. Niedrige Transaktionsgebühren**

Alle Plattformen verdienen einen Großteil ihres Umsatzes durch das Erheben von Transaktionsgebühren. Oft unterscheiden sich diese von Plattform zu Plattform. Ein Abwägen der Gebühren untereinander ist somit durchaus sinnvoll.

**3. Handeln von Kryptowährungen gegen Fiatgeld**

Das Handeln von Kryptowährungen gegen Fiatgeld (Euro, Dollar, usw.) ist von großem Vorteil. Denn so entfällt der oft mühsame Währungswechsel. Fehlt die Möglichkeit dazu, dann können Trader ausschließlich Kryptowährungen untereinander tauschen.

**4. Breite Auswahl an verschiedenen Kryptowährungen**

Eine breite Auswahl an handelbaren Kryptowährungen ist von großem Vorteil. Die Kurse verschiedener Kryptowährungen unterliegen unterschiedlich großen Schwankungen und manche Strategien funktionieren bei einigen Währungen besonders gut.

Anforderungen an die API:**1. Geeignete Dokumentation der API**

Um später mit den von der API bereitgestellten Funktionen arbeiten zu können, bedarf es einer guten und nutzerfreundlichen Dokumentation. Wird diese gar nicht oder in nur sehr geringem Maße von der Plattform angeboten, gestaltet sich die Einbindung und Verwendung der Funktionalitäten sehr schwierig und mühsam.

**2. Bereitstellung von Marktdaten**

Einer der wohl wichtigsten Anforderungen an eine API für Crypto-Trading ist die Bereitstellung von Marktdaten. Dabei müssen nicht nur aktuelle Preisdaten zur Verfügung gestellt werden, sondern auch historische - also bereits vergangene - Preisdaten bereitgestellt werden. Diese werden vor allem für den Backtest und die Verwendung von Strategien und deren Indikatoren benötigt.

**3. Bereitstellung von Symbolinformationen**

Als Symbolinformationen bezeichnet man Informationen über die von der Handelsplattform angebotenen Symbole bzw. Kryptowährungen. Ein Symbol bezeichnet die Art des Handels und besteht dabei immer aus den Währungen zwischen denen gehandelt wird - also beispielsweise Bitcoin und Euro. Natürlich möchte man wissen, welche Symbole derzeit handelbar sind und welche Kaufoptionen für sie möglich sind.

**4. Bereitstellung der Börsenhandelsregeln**

Alle Handelsaktionen unterliegen Handelsregeln. Dabei unterscheidet man zwischen Handelsregeln für die Börse und den Handelsregeln eines bestimm-

ten Symbols wie z.B. Mindestmenge und Maximalpreis.

#### 5. Platzieren von verschiedenen Kaufaufträgen

Grundlegend muss es möglich sein, Käufe und Verkäufe platzieren zu können. Für zukünftige Zwecke werden jedoch die gängigen Order Types wie beispielsweise der Stop-Loss- oder Limit-Order benötigt.

#### 6. Bereitstellung von Accountdaten

Natürlich soll es außerdem möglich sein, Accountdaten per API abfragen zu können. So können Daten wie z.B. derzeitiges Kapital, offene Trades und viele andere Dinge dargestellt oder visualisiert werden.

### 3.1.2. Schnittstelle der API

Eine von einer Plattform bereitgestellte API reicht natürlich nicht aus, um loslegen zu können. Zunächst muss sie sorgfältig in das System eingebunden werden, das sie nutzen möchte. Dazu muss eine weitere Schnittstelle entwickelt werden - sozusagen eine Schnittstelle der Schnittstelle, um die Nutzung der API einfacher zu gestalten. Die Schwierigkeit liegt darin, die Schnittstelle möglichst generisch und unkompliziert zu entwerfen. Es kann nämlich durchaus vorkommen, dass sich eine andere Plattform als besser erweist als die Bisherige, oder eine API nicht länger unterstützt wird. Somit ist es überaus wichtig, sicherzustellen, dass beim Austausch einer API so wenige andere Systemkomponenten beeinträchtigt werden wie möglich.

## 3.2. Algorithmische Handelsstrategien und technische Indikatoren

Die richtige Trading Strategie entscheidet über Gewinn und Verlust während technische Indikatoren die Werkzeuge jeder algorithmischen Strategie sind. Somit stellen beide also eine enorm wichtige Komponente des Systems dar. Wie bereits angedeutet, soll der Fokus jedoch weniger auf der Findung und Konzeption neuer Strategien und Indikatoren liegen, sondern darauf, wie diese möglichst einfach und modular in das System eingebunden werden können. Ein Blick ins World Wide Web genügt, um zu wissen, dass dort hunderte Kaufstrategien und Indikatoren auf eine Implementierung in Code warten. Welche Strategie davon nun die Beste ist, wird der Backtest früher oder später herausfinden. Das ist jedoch nicht Teil dieser Arbeit.

Teil dieser Arbeit ist es aber, das System auf die parallele Nutzung vieler Strategien und Indikatoren vorzubereiten, bzw. das System so zu entwerfen, dass diese nach Belieben ein- und ausgehängt werden können. Das bedeutet, dass der Backtest beispielsweise in der Lage sein soll, jede beliebige implementierte Strategie für seinen

Test zu Nutzen. Entscheidet man sich dazu, eine neue Strategie zu implementieren, soll die Strategie (inklusive ihren Indikatoren) ins System eingebunden werden können, ohne dabei die anderen Komponenten des Systems zu verändern - also nahtlos.

#### Anforderungen an die technischen Indikatoren:

1. **Berechnen der Indikatordaten für gegebene Marktdaten**

Jeder Indikator soll in der Lage sein, seine eigenen Indikatordaten auf der Basis gegebener Marktdaten zu berechnen. Bereitgestellt werden diese Daten durch die jeweilige Schnittstelle der Trading-Plattform- API.

2. **Anhängen der Indikatordaten an die gegebenen Marktdaten**

Außerdem soll es möglich sein, die zuvor berechneten Indikatordaten an die Marktdaten anzuhängen, um somit ein Datenobjekt zu bilden, welches der jeweiligen Trading Strategie von Nutzen ist.

#### Anforderungen an die Trading Strategie:

1. **Eigenständiges Hinzufügen der benötigten Indikatoren**

Jede Strategie benötigt eine mehr oder weniger große Anzahl an Indikatoren, durch welche die Strategie in der Lage ist, Kauf- und Verkaufsentscheidungen zu treffen. Jede Strategie sollte daher in der Lage sein, die benötigten Indikator-Instanzen selbst zu erzeugen und deren Daten anschließend an die Marktdaten anhängen zu lassen.

2. **Überprüfen der Buy Condition**

Die Buy Condition bezeichnet die Bedingung des Kaufes - also die Bedingung, die erfüllt sein muss, damit die Strategie einen Kauf für sinnvoll hält. Für die zukünftige Nutzung des Systems, soll die Strategie also die Möglichkeit besitzen, Echtzeitdaten zu überprüfen und für diese zu Entscheiden, ob ein Kauf platziert werden soll oder nicht.

3. **Erzeugung von Kaufsignalen**

Ist die Buy Condition einer Strategie erfüllt, muss dem System ein Kauf signalisiert werden. Dies soll durch die Erzeugung von sogenannten Buy Signals erreicht werden, die dem System die zum Kauf benötigten Informationen bereitstellen, womit dann ein Kauf platziert werden soll.

### 3.3. Backtesting von Trading Strategien

Der Backtest ist die wohl wichtigste Komponente des zu entwickelnden Systems. Durch den Backtest soll der Nutzer in der Lage sein, eine beliebige, im System implementierte, Trading Strategie auszuwählen und auf Trading Symbole, wie bspw. Bitcoin/Euro oder Ethereum/Euro laufen zu lassen. Da der Backtest für seinen Rückvergleich sowohl die Trading Strategie als auch die vergangenen Marktdaten benötigt, erfordert er das Zusammenspiel der anderen Komponenten mit sich selbst.

#### Anforderungen an den Backtest:

##### 1. **Durchführen des eigentlichen Backtest**

Zur Durchführung des eigentlichen Backtests gehört das Auswerten der historischen Marktdaten. Es soll möglich sein, ein Startkapital festzulegen, mit welchem der Backtest startet. Anschließend soll ein Echtzeitszenario nachgeahmt werden, bei dem der Backtest chronologisch durch die historischen Marktdaten geht und ein Kaufen und Verkaufen von Kryptowährungs-Instanzen simuliert.

##### 2. **Sammeln von Backtestdaten und Statistiken**

Um erkennen zu können, ob die Strategie innerhalb des Zeitraums der vergangenen Marktdaten Erfolg gehabt hätte, ist es wichtig, während des Backtests eine Vielzahl von Daten und Statistiken zu sammeln. So soll am Ende des Rückvergleichs beispielsweise ersichtlich werden, wie sich das Kapital entwickelt hat, wie viel Geld ausgegeben wurde und wie viel Gewinn die Strategie generiert hätte.

##### 3. **Bereitstellung der Backtestdaten**

Die während des Backtests gesammelten Informationen sollen dem Nutzer in einem gut lesbaren Format ausgegeben werden. Damit soll ihm die Möglichkeit gegeben werden, die Effizienz der Strategie anhand der gesammelten Statistiken zu bewerten.

##### 4. **Visualisierung der Backtestdaten**

Um dem Nutzer das Evaluieren der Backtest-Ergebnisse einfacher zu gestalten, sollen die historischen Marktdaten mithilfe eines Candlestick Charts visualisiert werden. Zusätzlich sollen auch die von der Strategie genutzten Indikatoren und die erzeugten Kauf- und Verkaufspunkte ersichtlich werden.

##### 5. **Speichern der Backtestdaten**

Damit der Nutzer die Ergebnisse verschiedener Backtests auch noch in Zukunft auswerten und vergleichen kann, sollen die Backtest-Ergebnisse persistent gemacht werden.

### 3.4. Entwurf und Verwaltung eines Trading Bots

Da innerhalb dieser Arbeit nur das Fundament für die Erstellung und Verwaltung von Trading Bots gelegt werden soll, wird der Trading Bot nicht in der Lage sein, eigenständig Einkäufe und Verkäufe zu tätigen. Vielmehr soll ein Konzept erarbeitet werden, das die Erstellung und Nutzung dieser Trading Bots in das System integriert. Die zukünftige Implementierung des Features für reale Käufe und Verkäufe soll dann auf diesem Konzept aufbauen.

#### Anforderungen an den Trading Bot

1. **Sammeln der nötigen Marktdaten**

Damit ein Trading Bot Anwendung findet, benötigt er einen Datensatz aus Echtzeitdaten. Daher soll er in der Lage sein, nach der Instanziierung eines neuen Bots, eigenständig die richtigen Marktdaten zu sammeln und zu verwalten.

2. **Updaten der Marktdaten**

Um die Marktdaten aktuell zu halten, muss der Trading Bot die Möglichkeit haben, seinen Datensatz selbständig zu aktualisieren.

3. **Evaluierung von Echtzeitdaten**

Der Bot soll zudem in der Lage sein, diesen ständig aktualisierten Marktdatensatz auszuwerten, um die Empfehlung eines Kaufes oder Verkaufes zu signalisieren.

#### Anforderungen an die Verwaltung des Trading Bots

1. **Hinzufügen und Entfernen neuer Trading Bots**

Die Verwaltungskomponente soll sowohl in der Lage sein alle Trading Bots zu verwalten, als auch neue zur Verwaltung hinzuzufügen.

2. **Starten und Stoppen der Trading Bots**

Analog zum Hinzufügen neuer Trading Bots, soll außerdem eine Möglichkeit zum Entfernen von Trading Bots gegeben sein.

Da der Trading Bot für die oben genannten Anforderungen sowohl Marktdaten als auch Trading Strategie zur Auswertung benötigt, muss eine reibungslose Zusammenarbeit mit den anderen Systemkomponenten gewährleistet sein.



### 3.5. Erstellung einer Konsolenanwendung

Alle bereits aufgeführten Komponenten und Features sollen Teil einer Konsolenanwendung werden, welche diese dann als System vereint und dem Nutzer zur Interaktion bereitstellt.

#### Anforderungen an die Konsolenanwendung

**1. Erstellen eines Backtests**

Die Erstellung eines Backtests soll dem Nutzer so einfach wie möglich gemacht werden. Der Nutzer soll daher eine Konfiguration des Backtests durchlaufen, indem er die verschiedenen Parameter wie Tradingsymbol oder Strategie, auswählt.

**2. Erstellen eines neuen Trading Bots**

Die Erstellung eines neuen Trading Bots soll ähnlich ablaufen wie die des Backtests. Auch hier soll der Nutzer eine Konfigurationsauswahl durchlaufen.

**3. Anzeigen der erstellten Trading Bots**

Sind erstmal ein paar Trading Bots erstellt, ist es natürlich von Vorteil sich diese Anzeigen zu lassen. Daher soll die Möglichkeit gegeben sein, alle vom System verwalteten Trading Bots aufzulisten und deren aktuellen Status anzuzeigen.

**4. Beenden des Programms**

Ein Beenden des Programms von der Konsolenanwendung aus soll ebenfalls bereitgestellt werden.

**5. Einfache und intuitive Bedienung**

Einer der wichtigsten Aspekte einer Benutzeroberfläche ist deren Bedienbarkeit. Kein Nutzer möchte sich bei der Nutzung eines Programms "verloren" fühlen und oder sich andauernd fragen, wie etwas funktioniert. Die Konsolenanwendung muss daher so gestaltet werden, dass sich auch Nutzer ohne Entwicklungshintergrund zurechtfinden.

**6. Übersichtliches und konsistentes Designkonzept**

Die Übersichtlichkeit und damit verbundene Konsistenz eines Designs innerhalb der Anwendung sind entscheidende Faktoren im Bereich Benutzerfreundlichkeit und sollten daher durchaus berücksichtigt werden.

### 3.6. Zusammenfassung

Die nachfolgende Abbildung (Abb. 3.1) dient der Veranschaulichung der zu entwickelnden Komponenten und deren Abhängigkeiten untereinander.

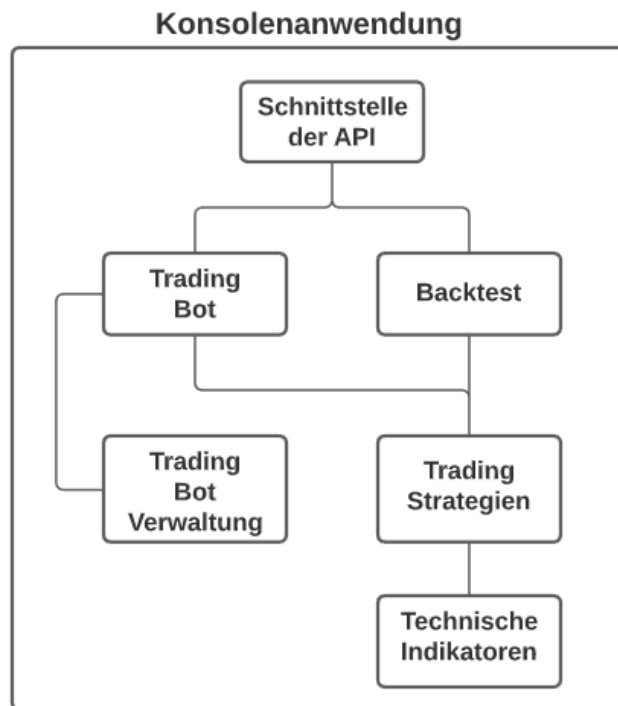


Abbildung 3.1.: Veranschaulichung der grundlegenden Komponenten

#### Erklärung der Abhängigkeiten:

- Sowohl Trading Bot als auch der Backtest sollen die Schnittstelle der API verwenden, um von dort aus die nötigen Marktdaten zu beziehen.
- Zusätzlich sollen Trading Bot und Backtest die im System zu implementierenden Trading Strategien nutzen, um damit deren gesammelten Marktdaten auszuwerten.
- Technische Indikatoren sind Bestandteil jeder Trading Strategie und sollen der Analyse von Marktdaten dienen.
- Instanzen eines Trading Bots sollen von einem Verwaltungssystem gehalten, gestartet und gestoppt werden.

## 4. Lösungskonzept

### 4.1. Schnittstelle für Handelsplattformen

#### 4.1.1. Auswahl einer Handelsplattform

Bevor mit der Konzeption einer Schnittstelle für eine Handelsplattform begonnen werden kann, muss sich für eine der vielen Plattformen entschieden werden. Diese Entscheidung fiel hier auf die Verwendung der Crypto-Trading-Plattform **Binance**. Die Plattform bietet den Handel von hunderten, verschiedenen Kryptowährungen an, welche auch gegen Fiatgeld gehandelt werden können. Zum Zeitpunkt dieser Arbeit berechnet Binance eine durchschnittliche Gebühr von etwa 0,1% auf jeden Trade, den ein Benutzer tätigt. Die Plattform gilt als sicher und unterstützt Zwei-Faktor-Authentifizierung. (vgl. [7])

Die von Binance bereitgestellte REST API ist, der Menge der im Internet bereitgestellten Lernressourcen nach, eine der verbreitetsten APIs unter algorithmischen Tradern. Zu ihr gibt es eine Vielzahl an Tutorials und eine ausgesprochen gute Dokumentation, welche auf GitHub unter folgendem Link zur Verfügung steht: <https://github.com/binance/binance-spot-api-docs/blob/master/rest-api.md>

#### 4.1.2. Entwurf einer Schnittstelle

Wie bereits angedeutet erfolgt die Kommunikation mit Binance über eine REST API. Das bedeutet, um an die Daten der Plattform zu kommen, müssen URLs erzeugt werden, mit deren Hilfe die sogenannten **Endpoints** kontaktiert werden. Jeder Endpoint erwartet eine unterschiedliche Menge an Parametern, die ebenso per URL übergeben werden müssen.

Die Aufgabe der Schnittstelle ist es dabei, den oben genannten Prozess innerhalb ihrer Methoden zu verstecken und wiederverwendbar zu machen. Das bedeutet, dass die anderen Komponenten des Systems nicht wissen müssen, wie die verschiedenen Endpunkte zu kontaktieren sind und wie sie deren Antworten verarbeiten müssen. Dargestellt wird dieses Konzept in Abbildung 4.1



Abbildung 4.1.: Konzept der REST API Schnittstelle

**Beispiel 1:**

Der Backtest benötigt eine Menge an Candlestick-Daten, an welche er theoretisch über die von Binance bereitgestellte REST API kommt. Dennoch tätigt der Backtest den API-Call nicht selbst, sondern lässt dies das Binance Interface für sich erledigen. Das bedeutet, der Backtest benötigt keinerlei Informationen darüber, wie er die API für die benötigten Daten kontaktieren muss, sondern erhält sie einfach über die Binance Schnittstelle.

**Beispiel 2:**

Nun benötigt auch ein Trading Bot eine Menge an Candlestick-Daten um seinen Echtzeit-Datensatz an Marktdaten zu initialisieren. Auch er muss nicht wissen, wie er an die Daten kommt, sondern lässt sie sich, ebenso wie der Backtest, von der Schnittstelle geben. Somit werden Arbeitsschritte voneinander getrennt und ebenso Code-Redundanz vermieden.

Um eine einfache Weiterverarbeitung der Candlestick-Daten zu gewährleisten, sollen die Daten in die, im Kapitel 2.4.1 beschriebene, Datenstruktur **DataFrame** eingepflegt werden und an die anderen Komponenten zurückgegeben werden.

## 4.2. Technische Indikatoren

Der technische Indikator dient der Strategie zur Analyse der Marktdaten. Damit diese der Strategie jedoch zur Verfügung stehen, muss die Möglichkeit bestehen, die Indikatordaten eines jeden Indikators auf gegebene Marktdaten zu berechnen und sie anschließend an diese anzuhängen. Somit genügt ein einfacher Erweiterungsprozess um die vom Indikator berechneten Daten aus, um die Datengrundlage einer jeden Trading Strategie zu bilden. Daraus ergibt sich folgender Entwurf (Abbildung 4.2)

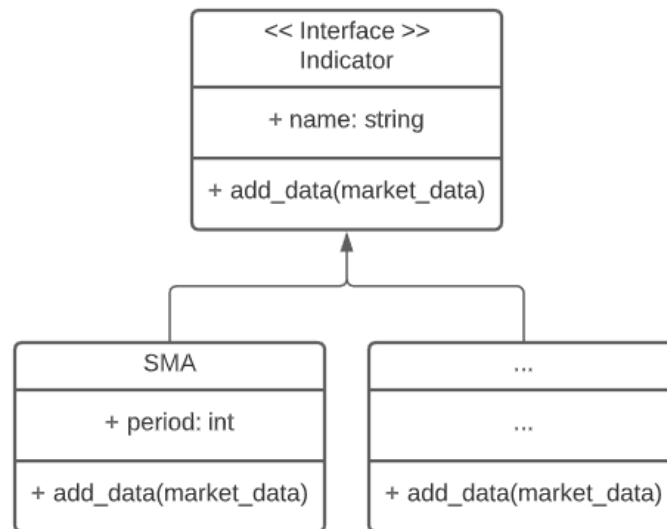


Abbildung 4.2.: Einfache Darstellung der Indikatoren durch ein Klassendiagramm

Durch die Definition eines Interface wird gewährleistet, dass jede neu hinzugefügte Klasse, die von der Klasse *Indicator* erbt, dazu verpflichtet ist, alle innerhalb des Interface definierten Methoden zu überschreiben. Damit wird erreicht, dass jeder Indikator das gleiche Funktionsprinzip hat und ein einheitliches Konzept bei der Implementierung verwendet wird.

Jeder Indikator-Instanz wird ein Name übergeben, welcher durch die Zuweisung an das Attribut *name* der eindeutigen Identifikation eines Indikators dient. Zusätzlich sind alle Indikatoren dazu verpflichtet, eine Methode `add_data(market_data)` zu implementieren, welche die indikatorspezifischen Daten an die übergebenen Marktdaten anhängt und zurückgibt.

Im Rahmen dieser Arbeit wird soll nur ein Indikator implementiert und verwendet werden - der **Simple Moving Average**, welcher bereits im Grundlagenkapitel 2.1.4 beschrieben wurde.

### 4.3. Algorithmische Trading Strategien

Eine gute Trading Strategie ist beim algorithmischen Traden der Schlüssel zum Erfolg, denn sie entscheidet darüber, ob ein Kauf platziert werden soll oder nicht. Wie sinnvoll dieser Kauf ist, obliegt der Entscheidung des Traders, welcher die Strategie implementiert. Damit im weiteren Verlauf des Projekts weitere, vermutlich bessere Strategien eingebunden werden können, ist der Entwurf eines guten Konzepts dafür von großer Wichtigkeit.

#### 4.3.1. Einbindung von Trading Strategien

Da jede Strategie eine gewisse Anzahl an bereitgestellten Funktionen bieten muss, macht die Nutzung eines Interfaces auch hier Sinn. Daraus folgt ein zu den Indikatoren ähnlicher Entwurf - dargestellt in Abbildung 4.3

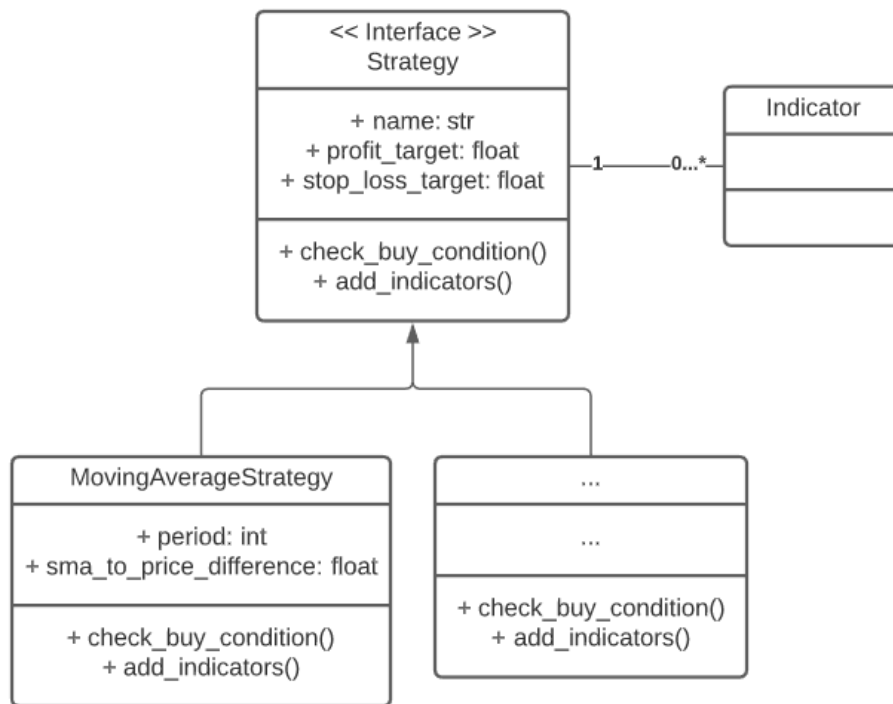


Abbildung 4.3.: Einfache Darstellung von Strategien durch ein Klassendiagramm

Wie aus dem Klassendiagramm ersichtlich wird, benötigt jede Strategie die Instanz-Attribute *name*, *profit\_target* und *stop\_loss\_target*. Letztere beiden werden zum Setzen von Stop-Loss Preisen im Backtest benötigt - mehr dazu im darauf folgenden Kapitel. Jede Strategie kann eine Vielzahl an verschiedener Indikatoren besitzen,

welche die Strategie zur Analyse der Daten herbeizieht (aus Gründen der Übersichtlichkeit hier als leere Klasse dargestellt). Jede neue implementierte Strategie erbt die Eigenschaften des Interface *Strategy* und verpflichtet sich damit, die im Interface definierten Methoden zu implementieren. *check\_buy\_condition()* wird dazu benötigt, die Buy Condition zu überprüfen, welche die Strategie ausmacht. Da jede Strategie selbst entscheiden soll, welche Indikatoren sie zur Analyse der Daten benötigt, soll sie durch die Methode *add\_indicators()* in der Lage sein, diese den Daten selbstständig hinzuzufügen.

Welche Strategie innerhalb dieses Projekts verwendet werden soll, lässt sich durch einen Blick auf das Klassendiagramm in Abbildung 4.3 vermuten. Ebenso wie der verwendete Indikator, wurde auch die **Moving Average Strategie** bereits im Grundlagenkapitel 2.1.5 beschrieben.

## 4.4. Backtest

### 4.4.1. Entwurf eines Backtests

Der Entwurf eines geeigneten Backtest stellt die wohl wichtigste Aufgabe dieses Kapitels dar und sollte gut durchdacht sein. Er steht in direkter Verbindung zu den bereits beschriebenen Klassen *Binance* und *Strategy*. Zusätzlich werden drei neue Klassen benötigt, um den Ablauf des Backtests möglich nah am Ablauf des realen Trading zu halten. In der nachfolgenden Abbildung (Abb. 4.4) wird dieses Konzept in vereinfachter Form dargestellt. Das heißt, es wurden im Hinblick auf die Übersichtlichkeit der Darstellung nur die wirklich für den Backtest benötigten Methoden und Attribute dargestellt.

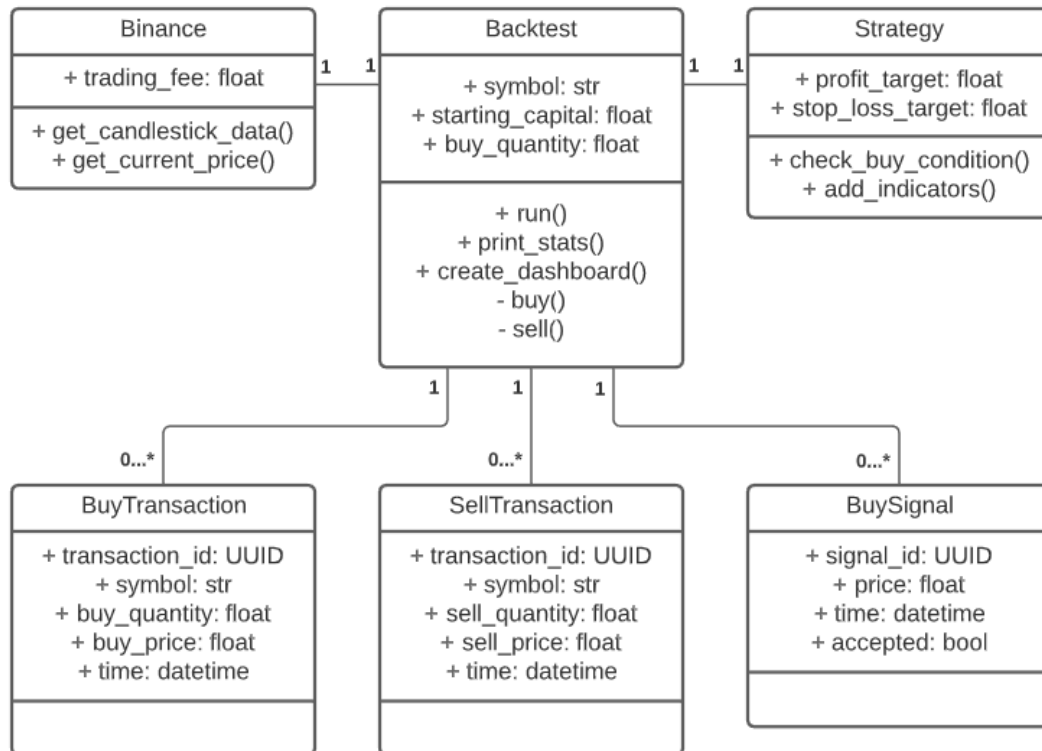


Abbildung 4.4.: Vereinfachte Darstellung der Klasse *Backtest* innerhalb eines Klassendiagramms

Jeder *Backtest* benötigt eine API *Binance*, von der er die benötigten Marktdaten abrufen kann. Diese Marktdaten müssen innerhalb des *Backtests* mithilfe einer *Strategy* verarbeitet werden. Um einen Kauf zu simulieren, benötigt der *Backtest* ein *BuySignal* welches ihm signalisiert, dass die Buy Condition der gewählten Strategie getroffen wurde. Die Klassen *BuyTransaction* und *SellTransaction* werden benötigt, um einen Kauf bzw. Verkauf zu simulieren und werden trotz ihrer Ähnlichkeit als zwei getrennte Klassen definiert, da die Nutzung einer einzelnen Transaktionsklasse innerhalb des *Backtests* schnell zur Verwirrung führen kann.

#### 4.4.2. Funktionsweise des Backtest

Interessant ist vor allem die Methode *run*, da sie für den automatischen Ablauf des *Backtests* zuständig ist und alle anderen Komponenten des *Backtests* und der oben definierten Klassen verwendet. Veranschaulicht wurde dieser Ablauf in Abbildung 4.5.



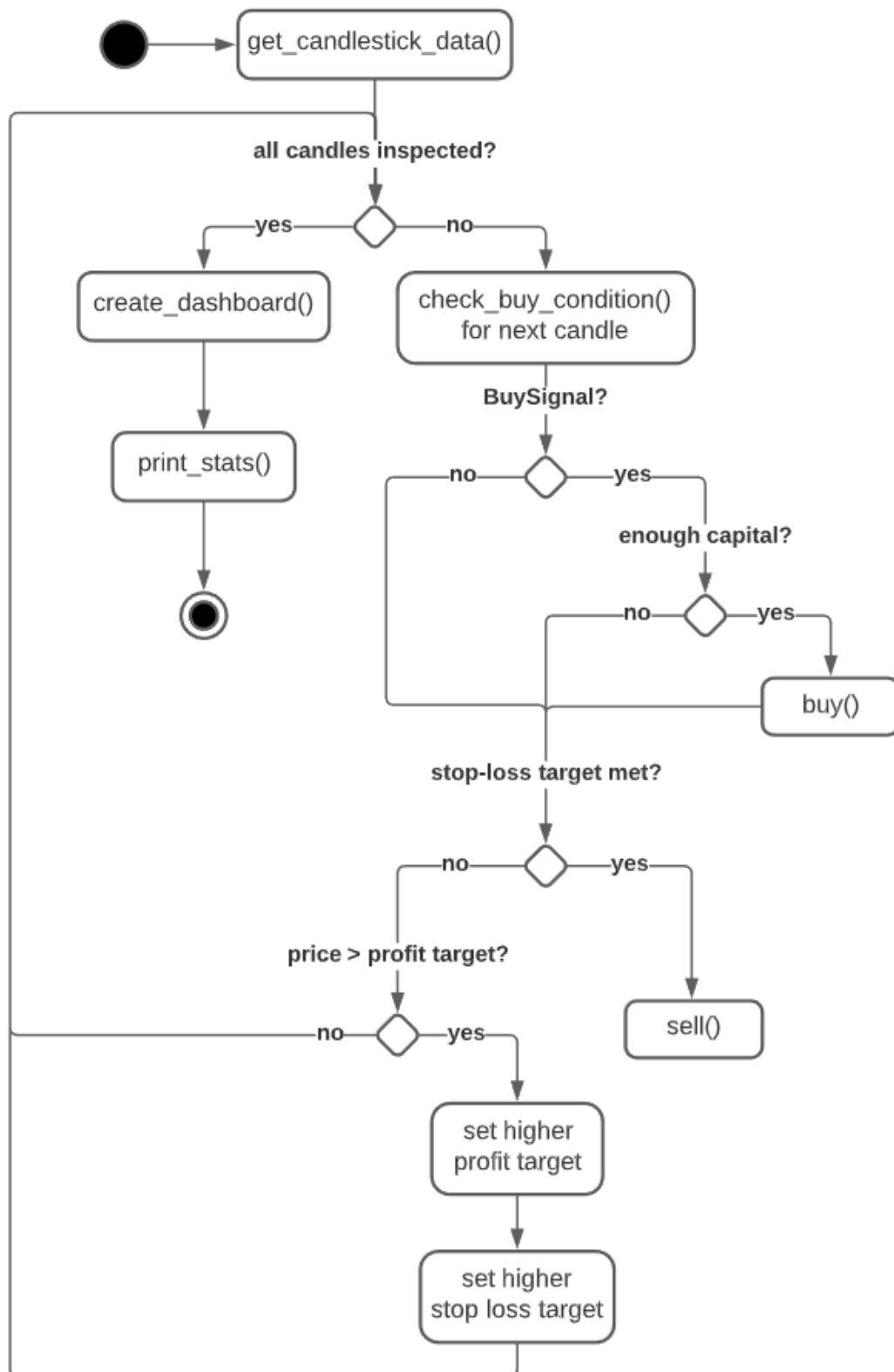


Abbildung 4.5.: Aktivitätsdiagramm für den Ablauf eines Backtest

**get\_candlestick\_data()**

Zu Beginn des eigentlichen Backtest wird die Schnittstelle *Binance* dazu beauftragt Candlestick-Daten von der REST API zu sammeln und in einem DataFrame zurück zu liefern. Dieser soll nun Candle für Candle analysiert werden.

**check\_buy\_condition()**

Für jede Candle lassen wir die gewählte Strategie ihre Buy Condition überprüfen. Erhält der Backtest ein *BuySignal* für die aktuelle Candle, wird überprüft, ob überhaupt genügend Kapital vorhanden ist, um einen Kauf zu platzieren.

**buy()**

Ist das für den Kauf benötigte Kapital vorhanden, platzieren wir einen Kauf. Innerhalb dessen soll die entsprechende *BuyTransaction* generiert und dem Backtest hinzugefügt werden. Nun sollte überprüft werden, ob der Preis das festgelegte *stop\_loss\_target* erreicht hat, welches ein Verkaufen von allen besessenen Coins bedeutet.

**sell()**

Analog für jede *BuyTransaction* soll nun eine *SellTransaction* generiert werden, welche einen symbolischen Kauf darstellt. Damit werden bei Erreichen des Stop-Loss Target alle bisher gehaltenen Coins verkauft.

**set higher profit target / set higher stop loss target**

Da wir einen Coin möglichst so lange halten möchten wie er profitabel ist, überprüft der Backtest nun, ob der Preis der aktuellen Candle das festgelegte *profit\_target* überschritten hat. Ist dies der Fall, legen wir höhere Werte für beide Variablen fest.

**create\_dashboard()**

Wurden alle Candles und damit die gesamten Marktdaten untersucht, wird ein Dashboard generiert, welches dem Nutzer eine Visualisierung der Marktdaten bietet.

**print\_stats()**

Während des gesamten Backtest werden verschiedene Daten zur Erhebung einer Statistik gesammelt, die dem Nutzer zeigt, wie erfolgreich der Backtest für eine gewählte Strategie und das gewählte Symbol ist.

**4.4.3. Visualisieren des Backtests**

Für die Erstellung eines Candlestick Chart o.ä. gibt es eine Vielzahl von Python Libraries, welche enorme Unterstützung bieten und einem den Großteil der eigentlichen Arbeit abnehmen. Die dabei verwendete Library nennt sich *plotly* und stellt nützliche Methoden für die Visualisierung von Finanzdaten bereit. Die erzeugten Diagramme lassen sich als HTML abspeichern und im Browser öffnen. Da jedoch

noch andere Daten mit in das Dashboard einfließen sollen, sollen mehrere HTML Objekte erzeugt und zu einer Datei zusammengefügt werden. Der Ablauf ist dabei wie in Abbildung 4.6 dargestellt.

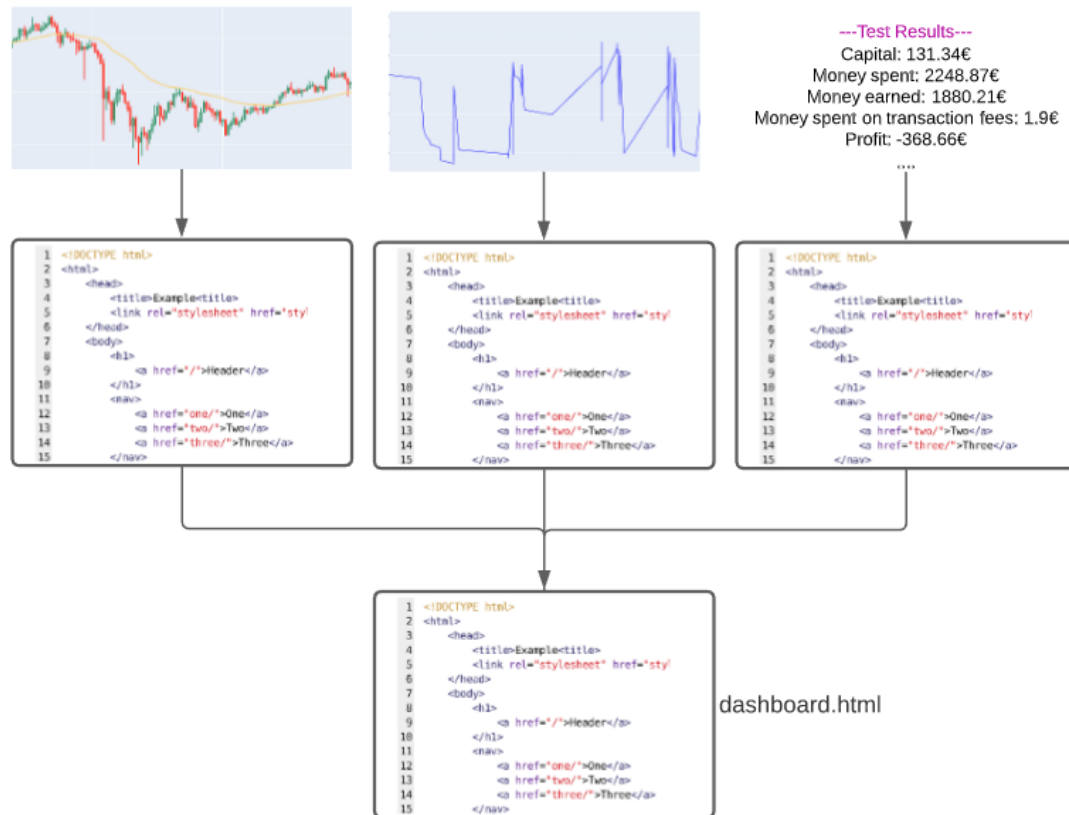


Abbildung 4.6.: Erstellungsvorgang eines Dashboards

- Erstellung eines Candlestick Charts als HTML Objekt:**  
Mithilfe der Library *plotly* wird ein Candestick Diagramm in Form eines HTML Objekts erzeugt und bis zur Weiterverarbeitung verwahrt.
- Erstellung eines Diagramm zur Visualisierung des Kapitals:**  
Innerhalb des Backtests wird bei jedem Kauf und Verkauf eine Veränderung des Kapitals zu diesem genauen Zeitpunkt der Transaktion gespeichert. Somit ist es möglich, das Kapital im zeitlichen Rahmen des Backtests als Liniendiagramm darzustellen. Dieses soll ebenfalls mithilfe von *plotly*, in Form eines HTML Objekts, erzeugt und verwahrt werden.
- Umwandlung der Backtest Statistiken in HTML Code:**  
Auch die beim Backtest generierten Statistikdaten sollen in das Dashboard eingebaut werden. Diese müssen mithilfe von Variablen-Substitution in vorbereiteten HTML Code eingefügt werden.

#### 4. Erzeugen des Dashboards:

Sind alle Objekte erzeugt, müssen diese noch zu einem großen Objekt zusammengefügt und zu einer Datei verarbeitet werden. Bei der Speicherung des HTML Dashboards wird gleichzeitig eine geeignete Ordnerstruktur erzeugt.

### 4.5. Entwurf und Verwaltung eines Trading Bot

Wie bereits erwähnt, wird im Rahmen dieser Arbeit kein funktionstüchtiger Trading Bot entworfen, sondern lediglich dessen Fundament und Integration im Trading System - dargestellt in Abbildung 4.7.

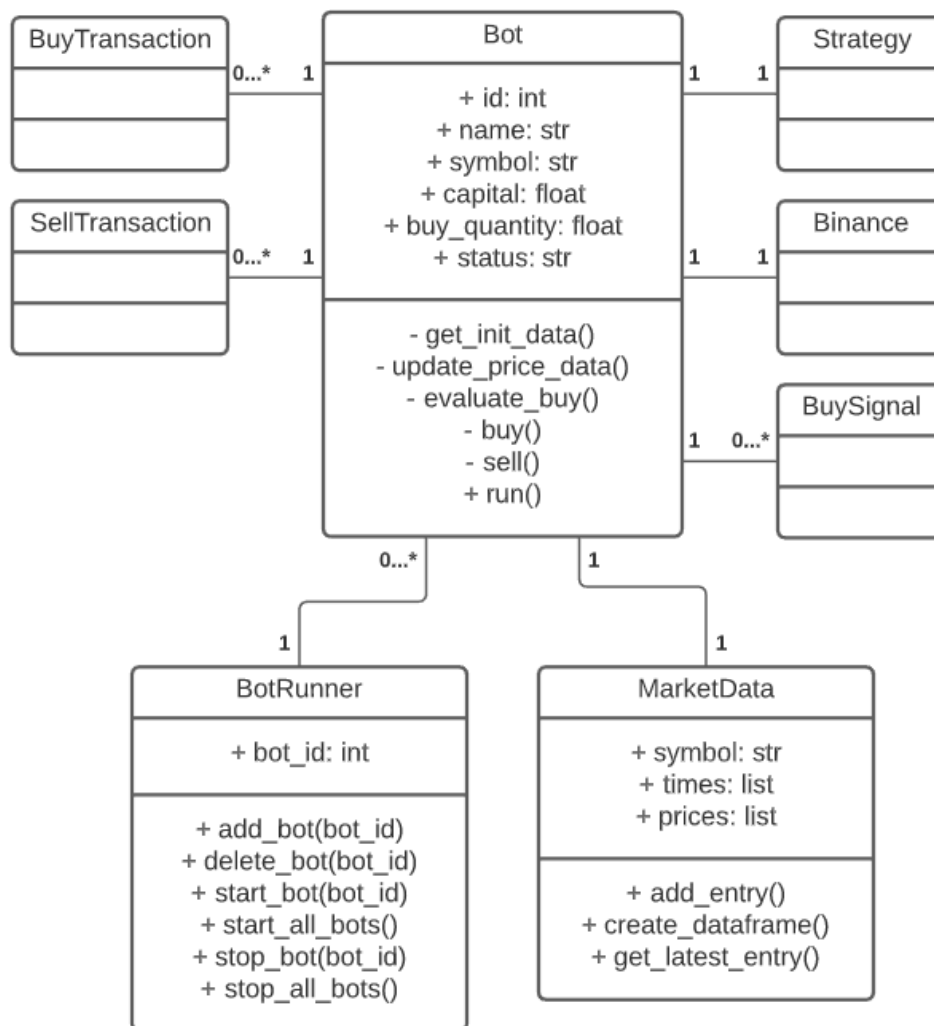


Abbildung 4.7.: Konzept eines Trading Bot und dessen Verwaltung als UML Diagramm

Ähnlich wie auch der Backtest, nutzt der Trading Bot nahezu alle anderen Komponenten des Systems. Die Schnittstelle *Binance* versorgt ihn mit Marktdaten, während die Strategie zur Auswertung dieser dient. Wird ein BuySignal erzeugt, entscheidet das Kapital über einen Kauf bzw. die Erstellung einer BuyTransaction. Analog dazu entscheidet das Stop-Loss Target der Strategie über einen Kauf bzw. die Erstellung einer SellTransaction.

Bisher unbekannt sind die Klassen *BotRunner* und *MarketData*. Die *MarketData* Klasse bildet ein Objekt aus Marktdaten, welches unter ständiger Verwaltung und Modifikation des Bots steht, da es eine feste Größe an beinhaltenden Daten hat und durchgehend aktuell gehalten werden muss.

Der BotRunner dient der Verwaltung aller Bots. Nach Erzeugung eines jeden Bots durch den Nutzer, wird dieser dem BotRunner hinzugefügt. Der BotRunner soll in der Lage sein, entweder alle oder einen einzelnen Bot zu starten oder zu stoppen. Somit bildet er zusammen mit der Konsolenanwendung die "höchste" Komponente des Systems.

## 4.6. Entwurf einer Konsolenanwendung

### 4.6.1. User Interface

Das User Interface ist das, was der User sieht, wenn er die Konsolenanwendung startet. Durch dieses ist er in der Lage, mit dem System zu interagieren und bestimmte Aktionen ausführen zu lassen. Das User Interface sollte sich möglichst intuitiv gestaltet werden um dem Nutzer eine einfache Nutzung zu bieten. Daher sollte auch das Designkonzept konsistent angewendet werden. Aus den bereits ermittelten Anforderungen ergibt sich das nachfolgende Designkonzept (Abb. 4.9).

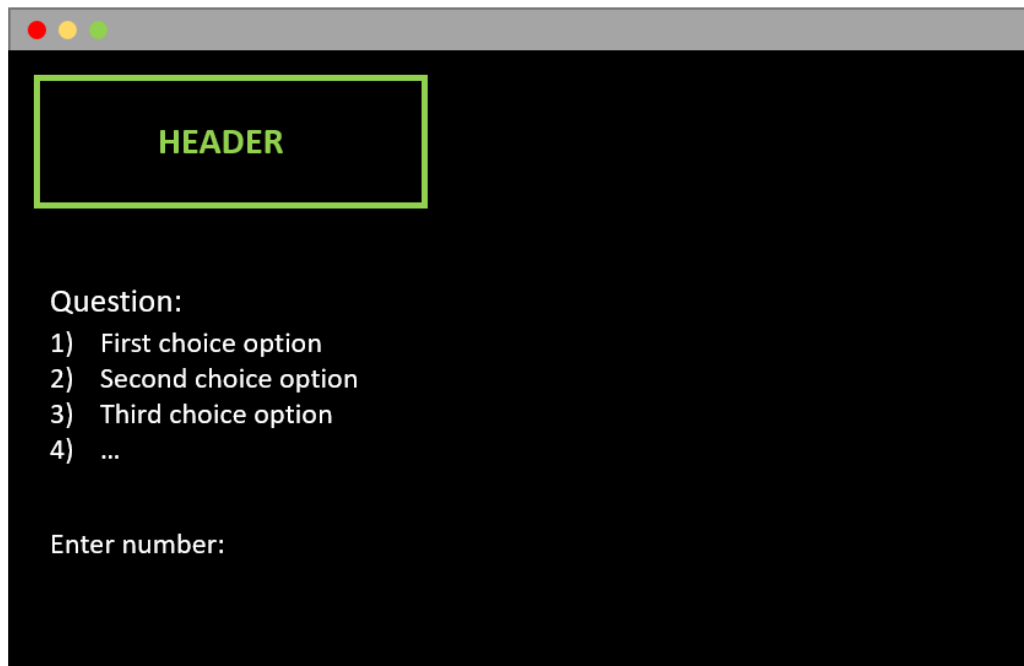


Abbildung 4.8.: User Interface der Konsolenanwendung

Bei jeder Aktion, die der Nutzer tätigen kann, soll ein dementsprechender Header angezeigt werden, der die gewählte Aktion beschreibt. So soll beim Starten der Anwendung beispielsweise ein Willkommens-Banner angezeigt werden, während beim Erstellen eines neuen Backtest der entsprechende Backtest-Header gezeigt werden soll.

Die Interaktion mit dem User soll hauptsächlich durch die Auswahl von Optionen geschehen. Dadurch wird der User bei den verschiedenen Aktionen an die Hand genommen und ihm immer alle möglichen Aktionen aufgezeigt. Startet er z.B. die Anwendung, soll der Nutzer gefragt werden, was er tun möchte. Zur Auswahl stehen die zur Verfügung stehenden Features, aus denen er wählen kann, indem er die korrespondierende Zahl eingibt.

#### 4.6.2. Zusammenspiel mit anderen Komponenten

Über das User Interface steuert der Nutzer alle großen Komponenten des Systems, welche durch den BotRunner, unter dem Dach der Konsolenanwendung, miteinander vereint werden (dargestellt in Abbildung 4.9). Backtest und Bot werden beide vom *CommandLineInterface* erstellt und im Falle des *Bot* an den *BotRunner* zur Verwaltung weitergereicht.

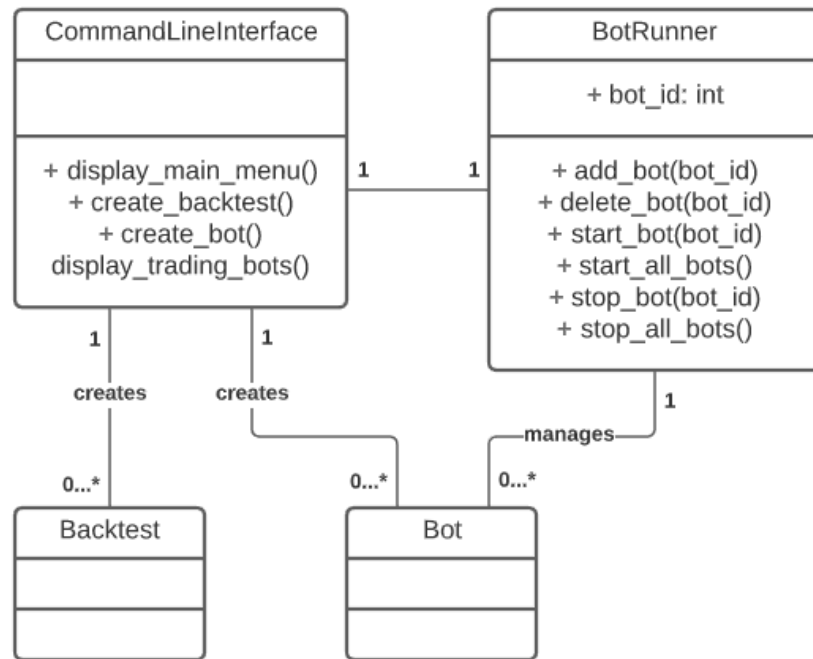


Abbildung 4.9.: Die Konsolenanwendung dargestellt als Klassendiagramm

## 5. Implementierung

### 5.1. Binance Schnittstelle

#### 5.1.1. Vorgehen

Im Prinzip sind alle Datenbeschaffungsfunktionen der Klasse *Binance* Implementierungen ähnlicher Manier. Das bedeutet, jede Methode, deren Aufgabe die Beschaffung eines gewissen Datensatzes, wie beispielsweise Preisdaten oder Symbolinformationen, folgt der gleichen Vorgehensweise und besteht immer aus den folgenden Schritten:

1. Erstellen der URL für den jeweiligen Endpoint der REST API
2. Senden einer HTTP Request an die erstellte URL
3. Überprüfung der empfangenen HTTP Response
4. Überführung der Daten in eine geeignete Form oder Datenstruktur

#### 5.1.2. Klassendefinition

Die Schnittstelle der Binance API wurde innerhalb einer Python File mit dem Namen *binance.py* implementiert. Hier wurde zunächst eine Klasse *Binance* definiert:

##### **binance.py**

---

```
1 class Binance:
2     # Symbols
3     SYMBOL_BITCOIN_EURO = "BTCEUR"
4     SYMBOL_ETHEREUM_EURO = "ETHEUR"
5     SYMBOL_LITECOIN_EURO = "LTCEUR"
6
7     # Endpoints
8     ENDPOINT_KLINES = "/api/v3/klines"
9     ENDPOINT_PRICE = "/api/v3/ticker/price"
10    ENDPOINT_TIME = "/api/v3/time"
11    ENDPOINT_TEST_ORDER = "/api/v3/order/test"
12    ENDPOINT_EXCHANGE_INFO = "/api/v3/exchangeInfo"
```



```
13
14 # Constructor
15 def __init__(self):
16     self.base = "https://api.binance.com"
17     self.trading_fee = Binance.TRADING_FEE
```

---

Quelltext 5.1: Binance Klassendefinition

Bevor die Klasse durch den Konstruktor instanziiert werden kann, werden mehrere Klassenvariablen definiert:

- **Symbols (Zeile 3-5):**  
Zunächst werden die für diese Arbeit verfügbaren Trading Symbole festgelegt. Jedes Symbol stellt die zwei Währungen dar, zwischen denen gehandelt werden soll.
- **Endpoints (Zeile 8-12):**  
Als nächstes werden die verschiedenen Endpunkte der REST API definiert, welche benötigt werden um die gewollten Informationen anzufragen.
- **Konstruktor (Zeile 15-17):**  
Anschließend wird ein Konstruktor implementiert, welcher die Klasse instanziiert. Dort werden die beiden Instanzvariablen *base* (URL der REST API) und *trading\_fee* (Prozentsatz der Trading-Gebühren) festgelegt.

### 5.1.3. API-Anfragen

Bevor die Beschaffung der Marktdaten o.ä. betrachtet werden kann, muss verstanden werden, wie eine Anfrage an die API der Trading-Plattform Binance gehandhabt wird.

#### binance.py

```
1 def http_request(self, endpoint, params=None):
2     # Create URL
3     url: str = self.base + endpoint
4     if params:
5         for i in range(len(params)):
6             if i == 0:
7                 # First param has to connect with a question mark to the url
8                 url = url + "?" + params[i]
9             else:
10                # Other params connect with a ampersand
11                url = url + "&" + params[i]
```

---

Quelltext 5.2: Binance API Calls

Eine URL wird erzeugt, indem man die *base* mit dem jeweiligen *endpoint* verbindet. Jede URL kann das Verhalten der API durch eine Mitgabe an Parametern modifizieren. Parameter werden, wie der Endpoint, beim Aufruf der Methode übergeben. Bei den Parametern geschieht dies über den optionalen Übergabewert *params*.

Ob Parameter übergeben werden oder nicht, wird in Zeile 4 abgefragt. Wenn ja, muss der erste Parameter mit einem **?** an die URL gehängt werden, während alle weiteren Parameter mit einem **&** hinzugefügt werden. So entsteht beispielsweise folgende URL: `https://api.binance.com/api/v3/ticker/price?symbol=LTCEUR`

#### **binance.py**

---

```
1 # Call url to get expected response
2 try:
3     response: Response = requests.get(url)
4 except requests.exceptions.ConnectionError as e:
5     logger.error(f"ConnectionError: {e}")
6     return False
7
8 # Check response
9 try:
10    response.raise_for_status()
11 except requests.exceptions.HTTPError as e:
12    logger.error(f"HTTP error: {e}")
13    return False
14
15 # Decode data
16 try:
17    data = response.json()
18 except JSONDecodeError as e:
19    logger.error(f"Could not decode data from {url}: {e}")
20    return False
21 else:
22    return data
```

---

Quelltext 5.3: Binance API Calls Fortsetzung

Nachdem die URL des gewollten Endpoints erzeugt wurde, werden die Daten nun abgefragt, überprüft und verarbeitet:

- **API-Aufruf (Zeile 2-6):** Um das Arbeiten mit HTTP Anfragen möglichst einfach zu gestalten wird das Package *requests* verwendet. Es wird eine GET Request mit der erzeugten URL gestartet, durch welche die API eine HTTP Response erzeugt, welche im besten Fall die gefragten Daten beinhalten. Sollte während der Anfrage ein Verbindungsfehler auftreten, wird dieser abgefangen und auf dem Terminal ausgegeben. Anschließend gibt die Methode *False* zurück, um dem Aufrufer zu signalisieren, dass etwas falsch gelaufen ist.
- **Verarbeiten der HTTP Response (Zeile 9-13):**  
Trat kein Verbindungsfehler auf, untersuchen wir die HTTP Request nun auf

seinen Status-Code. Dieser gibt Auskunft darüber, ob alles geklappt hat, oder Fehler aufgetreten sind. Wird ein fehlerhafter Status-Code entdeckt, wird ein Fehler erzeugt, welcher in gleicher Manier abgefangen wird wie zuvor der Verbindungsfehler.

- **Dekodieren der Daten (Zeile 16-22):**

Da die Daten einem textbasierten Format namens JavaScript Object Notation (JSON) von der API an den Aufrufer zurückgegeben werden, müssen diese vor Verwendung noch dekodiert werden. Dabei werden die in der Response enthaltenen Daten durch das Package *json* automatisch dekodiert. Tritt dabei ein Fehler auf, wird der Fehler wie bereits bekannt abgefangen und *False* zurückgegeben. Läuft alles wie es soll, werden die Daten in einer geeigneten Datenstruktur zurückgegeben.

#### 5.1.4. Sammeln von Candlestick-Daten

Eine der wichtigsten Funktionen der Binance Schnittstelle ist das Sammeln von Candlestick-Daten, welche sowohl im Backtest als auch beim Initialisieren des Trading Bots benötigt werden.

Um dieser Anforderung Genüge zu tun, wurde eine Klassenmethode *get\_candlestick\_data()* implementiert, welche eine festgelegte Menge an Candles für ein bestimmtes Trading Symbol von der API anfordert und verarbeitet.

##### **binance.py**

---

```

1  def get_candlestick_data(self, symbol, interval="1h", end_time=None,
    limit=1000):
2      # Get data
3      params = [
4          "symbol=" + symbol,
5          "interval=" + interval,
6          "limit=" + str(limit)
7      ]
8      if end_time:
9          params.append("endTime=" + str(end_time))
10     data = self.http_request(endpoint=self.ENDPOINT_KLINES, params=params)
11     if not data:
12         logger.error("Missing candlestick data")
13     return False

```

---

Quelltext 5.4: Sammeln von Candlestick-Daten

Erklärung der Übergabeparameter:

- **symbol:**

Das Symbol bzw. die Währungen, für welche die Candlestick-Daten angefragt

werden.

- **interval="1h":**

Dieser optionale Parameter legt das Zeitintervall fest, welches die angefragten Kerzen haben sollen. Diese gibt es beispielsweise in Minuten, Stunden, Tagen und noch mehr. Wird kein Wert für den Parameter übergeben, wird ein Standardwert von einer Stunde pro Intervall festgelegt.

- **end\_time=None:**

Ein ebenfalls optionaler Parameter, welcher eine Zeit festlegt, von welchem aus die Daten (rückwärtsgehend) geholt werden soll. Wird kein Wert übergeben, werden einfach die neuesten Kerzen geholt.

- **limit=1000:**

Das Limit legt fest, wie viele Kerzen abgefragt werden sollen. Dieses wird von Binance festgelegt und beträgt standardmäßig 500, maximal jedoch 1000.

In Zeile 2-10 werden die für die Candlestick-Daten benötigten Parameter festgelegt, woraufhin anschließend die Daten geholt werden. Gab es während des Datenbeschaffungsprozess ein Problem, erhalten wir den boolschen Wert *false*. Daher überprüfen wir in Zeile 11, ob beim Beschaffen der Daten alles geklappt hat. Sollte dies nicht der Fall sein, wird der Fehler auch in dieser Methode geloggt und *False* zurückgegeben. Erhalten wir jedoch die gewollten Daten, liegen diese nun in folgendem Format vor:

---

```

1  [
2    [
3      1499040000000,      // Open time
4      "0.01634790",      // Open
5      "0.80000000",      // High
6      "0.01575800",      // Low
7      "0.01577100",      // Close
8      "148976.11427815", // Volume
9      1499644799999,      // Close time
10     "2434.19055334",    // Quote asset volume
11     308,                // Number of trades
12     "1756.87402397",    // Taker buy base asset volume
13     "28.46694368",      // Taker buy quote asset volume
14     "17928899.62484339" // Ignore .
15   ],
16   [
17     ...
18     ...
19     ...
20   ],
21   ...
22 ]
```

---

Quelltext 5.5: Candlestick-Daten in Rohform

Die hier abgebildete Datenstruktur *list* enthält alle Candles in Form von weiteren Lists - sozusagen ein zweidimensionales Array. Innerhalb jeder Candle befinden sich die oben erkennbaren Daten wie *Open time* und *Open*. Diese sollen nun für eine bessere Weiterverarbeitung in die Datenstruktur DataFrame eingepflegt werden:

#### binance.py

---

```
1 df = DataFrame(data)
2 df = df.drop(range(6, 12), axis=1)
3 col_names = ["time", "open", "high", "low", "close", "volume"]
4 df.columns = col_names
5
6 # Transform values from strings to floats
7 for col in col_names:
8     df[col] = df[col].astype(float)
9 return df
```

---

Quelltext 5.6: Sammeln von Candlestick-Daten Fortsetzung

Erklärung des restlichen Code:

- **Erstellen eines DataFrame (Zeile 1-4):**

Das DataFrame Objekt wird erzeugt, indem man ihm die Candlestick-Daten in der bereits vorliegenden Form einer Liste/Array übergibt. Anschließend werden die letzten sechs Spalten (beginnend bei "Close time") entfernt, da diese für die hier vorliegenden Zwecke nicht benötigt werden.

- **Transformieren der Textwerte in Gleitkommawerte (Zeile 7-8):**

Um die in den Kerzen enthaltenen Werte Zahlenwerte für arithmetische Dinge nutzen zu können, müssen sie vorerst in Gleitkommawerte umgewandelt werden. Anschließend wird der DataFrame an den Aufrufer der Methode zurückgegeben.

Jede Zeile des DataFrame stellt nun eine Kerze inklusive ihren korrespondierenden Daten dar. Abbildung 5.1 zeigt einen solchen DataFrame und mehrere dieser Kerzen.

	time	open	high	low	close	volume
0	161176680...	25309.28000	25520.63000	24917.28000	25233.84000	227.25194
1	161177040...	25233.84000	25244.04000	24410.28000	24586.94000	287.10042
2	161177400...	24584.13000	25500.00000	24575.59000	25393.57000	351.69656
3	161177760...	25403.45000	26137.81000	25234.02000	26083.28000	369.65110
4	161178120...	26097.06000	26182.00000	25407.29000	25580.54000	197.83394
5	161178480...	25580.49000	25660.99000	25123.58000	25279.97000	118.46124
6	161178840...	25279.97000	25430.00000	24953.38000	25159.79000	236.69100
7	161179200...	25159.79000	25480.88000	24769.02000	25441.39000	209.33883
8	161179560...	25450.68000	25580.38000	25175.65000	25569.55000	125.42863
9	161179920...	25573.48000	25837.46000	25502.70000	25696.23000	119.68276
10	161180280...	25698.18000	26108.26000	25643.69000	26018.65000	88.13577
11	161180640...	26023.69000	26160.22000	25740.00000	25928.12000	61.07358
12	161181000...	25935.13000	26308.03000	25740.08000	26246.57000	82.33762
13	161181360...	26244.96000	26364.00000	26041.30000	26073.46000	76.80744
14	161181720...	26078.95000	26108.45000	25729.48000	25809.16000	136.99568
15	161182080...	25810.04000	25924.19000	25668.11000	25756.85000	157.76159
16	161182440...	25757.26000	26057.71000	25550.00000	26041.40000	137.35817
17	161182800...	26044.70000	26125.00000	25691.49000	26036.96000	141.63390
18	161183160...	26044.10000	26140.23000	25831.75000	26029.06000	111.02359

Abbildung 5.1.: Candlestick-Daten innerhalb eines DataFrame

## 5.2. Indikatoren

### 5.2.1. Abstrakte Basisklasse Indicator

Interfaces werden in Python anders gehandhabt als in den meisten anderen Sprachen und können in ihrer Designkomplexität variieren. Klassische Interfaces, wie man sie aus Sprachen wie Java oder C++ kennt, existieren in Python nicht. Daher wird auf die Verwendung einer Python Library mit dem Namen Abstract Base Class (ABC) verwendet. Durch dies ist man in der Lage, definierte Methoden als *abstract* zu deklarieren, was den gewünschten Effekt eines Interfaces erzielt.

Da die Basisklasse der Indikatoren keine wirkliche Funktionalität enthält, besitzt sie eine simple Implementierung, dargestellt im nachfolgenden Code-Ausschnitt:

#### indicators.py

```

1 from abc import ABC, abstractmethod
2
3 class Indicator(ABC):
4     def __init__(self, name: str) -> None:
5         self.name: str = name

```

```

6
7     @abstractmethod
8     def add_data(self, data: DataFrame, column_name: str):
9         # Needs to be overridden in every subclass
10        raise NotImplementedError("Missing implementation: Please
           override this method in the subclass")

```

---

Quelltext 5.7: Implementierung der abstrakten Basisklasse *Indicator*

Durch das Erben von der Klasse *ABC* in Zeile 3 wird die Verwendung der Library *ABC* ersichtlich. Anschließend wird in Zeile 4-5 ein Konstruktor erzeugt, welche einen Indikator mit einem Namen instanziiert.

Der wichtigere Teil befindet sich in Zeile 7-10. Wie bereits in Kapitel 4.2 beschrieben, soll jeder Indikator dazu verpflichtet sein, die Methode *add\_data* zu implementieren. Dies wird erreicht, indem über der Methodendeklaration ein sogenannter Decorator *@abstractmethod* platziert wird (Zeile 7). Um dem Entwickler neuer Indikatoren ein Fehlverhalten im Falle einer Nicht-Implementierung mitzuteilen, erschaffen wir einen *NotImplementedError* in Zeile 10. Wird nun ein neuer Indikator implementiert, die Methode *add\_data* jedoch nicht überschrieben, erhebt das System einen Laufzeitfehler.

### 5.2.2. Simple Moving Average

Für viele technische Indikatoren gibt es bereits bereitgestellte Implementierungen, welche durch die Verwendung von Packages in das System integriert werden können. Auch wenn diese als bereits fertige Objekte importiert werden können, soll die klare Struktur der Indikatoren beibehalten werden. Daher wird zunächst die Klasse *SimpleMovingAverage* definiert:

#### **indicators.py**

```

1  from pyti.smoothed_moving_average import smoothed_moving_average as sma
2
3  class SimpleMovingAverage(Indicator):
4
5      def __init__(self, name, period):
6          super().__init__(name) # Init parent class
7          self.period: int = period

```

---

Quelltext 5.8: Klassendefinition *SimpleMovingAverage*

Durch das Erben von der Klasse *Indicator* in Zeile 3, wird die Verwendung der zuvor definierten abstrakten Basisklasse deklariert. Im Konstruktor in Zeile 5-7 wird die Instanzvariable der Basisklasse *Indicator* sowie die Instanzvariable *period* initialisiert.

Der wichtige Teil passiert jedoch in der Implementierung der nun überschriebenen Methode *add\_data*:

**indicators.py**

---

```
1 def add_data(self, data, column_name):
2     calc_column = data[column_name].tolist()
3     sma = sma(calc_column, self.period)
4     data[self.name] = sma
5     return data
```

---

Quelltext 5.9: Implementierung der Methode *add\_data*

Beim Aufruf der Methode werden zwei Übergabeparameter erwartet:

- **data:** Der (Candlestick) DataFrame, an welchen die Indikatordaten angehängt werden sollen.
- **column\_name:** Der Name der Spalte, auf deren Datengrundlage der Indikator berechnet werden soll.

Zunächst werden die Daten der Spalte, auf deren Grundlage der Indikator berechnet werden soll, aus dem DataFrame extrahiert und zu einer Liste, bestehend aus diesen Werten, transformiert (Zeile 2). Anschließend wird unter der Verwendung dieser Daten sowie der festgelegten Periode ein SimpleMovingAverage erzeugt (Zeile 3). Am Ende werden die erzeugten Daten an den DataFrame in Form einer neuen Spalte angehängt und zurückgegeben.

Da man sich dieses Prinzip ohne ein wenig Übung im Umgang mit den genannten Datenstrukturen vielleicht schwer vorstellen kann, folgt ein visuelles Beispiel in Abbildung 5.2. In diesem Beispiel wird der Simple Moving Average Indikator für gegebene Candlestick-Daten berechnet und an diese angehängt.



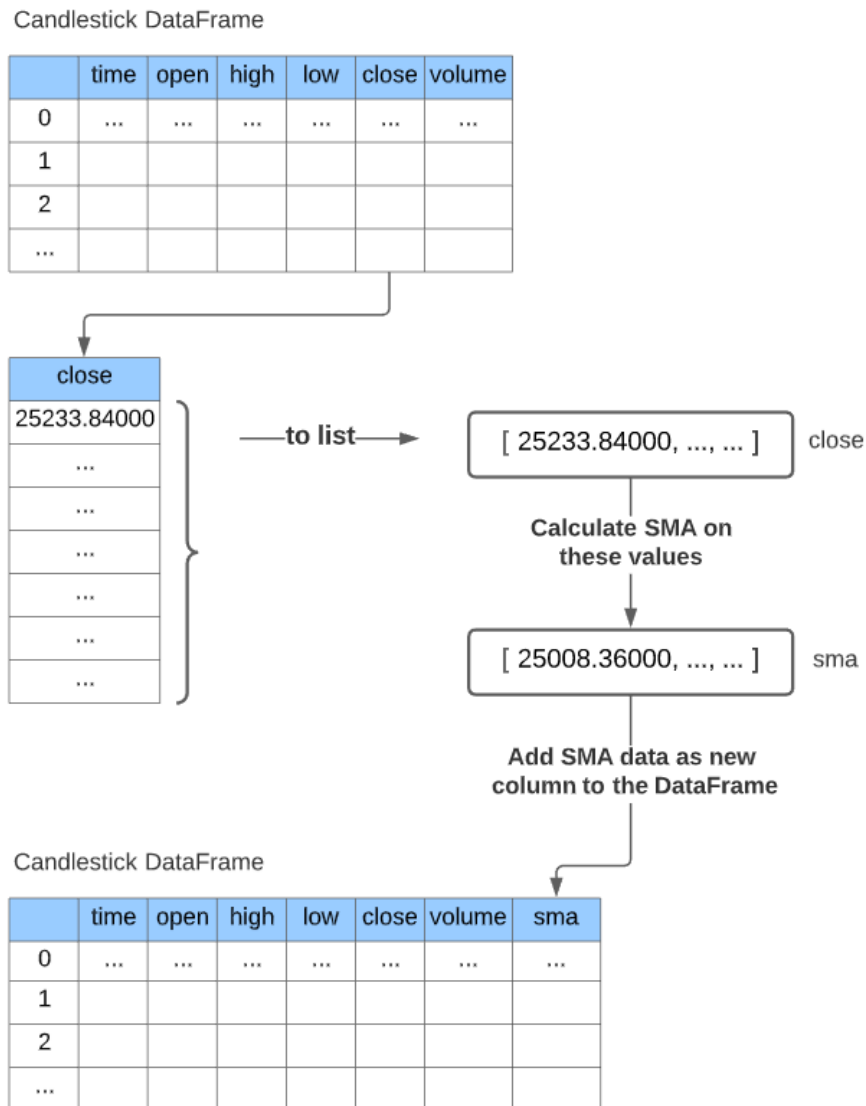


Abbildung 5.2.: Berechnen und Hinzufuegen des SMA Indikator an einen DataFrame

### 5.3. Moving Average Strategie

Die eigentliche Implementierung der Moving Average Strategie ist recht simpel. Da die Verwendung einer abstrakten Basisklasse schon im vorangegangenen Kapitel aufgezeigt wurde, wird diese Information hier weggelassen.

Zunächst wird die Klasse definiert und ein paar wichtige Instanzvariablen initialisiert:

**indicators.py**


---

```

1 class MovingAverageStrategy(Strategy):
2
3     def __init__(self):
4         self.name = "Moving Average Strategy"
5         self.profit_target = 1.05
6         self.stop_loss_target = 0.85
7         self.sma_to_price_difference = 1.03
8         self.indicators = list()

```

---

Quelltext 5.10: Klassendefinition der Moving Average Strategy

Erklärung der Instanzvariablen:

- **name:** Name der Strategie
- **profit\_target:** Das Profit Ziel liegt bei 5% Preiszuwachs
- **stop\_loss\_target:** Das Stop-Loss Target liegt bei 15% unter dem Kaufpreis. Wird dieser Preis erreicht, wird verkauft.
- **sma\_to\_price\_difference:** Liegt der Preis 3% unter dem Wert des Simple Moving Average, soll gekauft werden
- **indicators:** Liste der von der Strategie verwendeten Indikatoren

Die Methoden der Strategie sind sehr einfach gehalten. Die Überprüfung der Daten zur Bestimmung eines Kaufes oder Nicht-Kaufes wird in gerade einmal 4 Zeilen Code erreicht:

**indicators.py**


---

```

1     def check_buy_condition(self, price, time, row=None):
2         sma = getattr(row, "slow_sma")
3
4         # Check if we meet our strategy condition
5         if sma > self.sma_to_price_difference * price:
6             return BuySignal(price, time)
7         else:
8             return False

```

---

Quelltext 5.11: Überprüfen der Kaufbedingung

In Zeile 5 wird überprüft, ob der Wert des SMA drei Prozent über dem zugehörigen Preis liegt. Ist dies der Fall, wird ein *BuySignal* erzeugt und zurückgegeben, ansonsten *False*.

Das automatische Hinzufügen von Indikatoren läuft ebenfalls sehr simpel ab. Dabei werden die jeweiligen benötigten Indikatoren instanziiert und deren *add\_data()*

Funktion verwendet, um dem DataFrame die Indikatordaten hinzuzufügen. Anschließend werden alle Indikatorinstanzen zur Liste der Indikatoren (*indicators*) hinzugefügt.

## 5.4. Backtest

Dass der Backtest die bisher komplexeste Komponente des Systems ist, spiegelt sich in seiner Implementierung wieder. Implementiert wurde der Backtest innerhalb der Datei *backtest.py*, welche mit über 400 Zeilen Code die größte Sourcefile darstellt.

### 5.4.1. Ablauf des Backtests

Das Herzstück des Backtests ist seine *run()* Methode. Diese definiert den kompletten Ablauf des Backtests, welcher in Abbildung 4.5 durch ein Aktivitätenprogramm grob beschrieben wird. Der Übersichtlichkeit zuliebe werden nur Ausschnitte in vereinfachter Form gezeigt:

#### **backtest.py**

---

```

1  profit_target_price = -1
2  stop_loss_price = -1
3  for index, candle in candlestick_data_frame.iterrows():
4      # Get close price and time of current candle
5      close_price = getattr(candle, "close")
6      time = getattr(candle, "time")
7      buy_signal = strategy.check_buy_condition(close_price, time, candle)
8
9      # Check whether we can buy
10     if buy_signal:
11         if capital >= close_price:
12             buy(buy_signal)
13
14         # We now have coins -> init targets
15         profit_target_price = close_price * strategy.profit_target
16         stop_loss_price = close_price * strategy.stop_loss_target
17
18     # Check wheter we can sell
19     if kept_coins:
20         low_price: getattr(candle, "low")
21
22         # Check whether stop-loss price was reached
23         if low_price <= stop_loss_price:
24             sell(stop_loss_price) # sell all coins for sl price
25             profit_target_price = -1 # reset
26             stop_loss_price = -1 # reset
27
28     # Check whether we need to set new targets

```

```

29     elif close_price >= profit_target_price != -1:
30         profit_target_price = close_price * strategy.profit_target
31         stop_loss_price = close_price * strategy.stop_loss_target
32
33     # Create dashboard
34     create_folder_structure()
35     create_candlestick_figure()
36     figures_to_html()
37     stats_to_html()
38     create_html_dashboard()
39     print_stats()

```

---

Quelltext 5.12: Vereinfachte Darstellung der Methode *run()*

Beim Ablauf des Backtests muss jede einzelne Kerze auf einen Kauf oder Verkauf überprüft werden. Für jede Kerze werden der Close Price und dessen korrespondierende Zeit ermittelt (**Zeile 5-6**). Diese werden nun der Strategie übergeben, welche überprüft, ob gekauft werden soll oder nicht (**Zeile 7**). Wurde ein *BuySignal* erzeugt, wird mithilfe von diesem ein Kauf simuliert (**Zeile 10-12**). Mithilfe des Preises, zu welchem gerade Coins erworben wurden, werden nun das Profitziel und der Stop-Loss gesetzt (**Zeile 15-16**).

Befinden sich nun Instanzen einer Kryptowährung im Besitz, muss auch für jede Kerze überprüft werden, ob der niedrigste Preis innerhalb ihres Zeitintervalls den definierten Stop-Loss Preis erreicht hat (**Zeile 19-23**). Ist dies der Fall, werden alle besessenen Coins verkauft und *profit\_target\_price* sowie *stop\_loss\_price* zurückgesetzt (**Zeile 24-26**).

Am Ende der Schleife wird überprüft, ob der Schlusspreis der aktuellen Kerze das definierte Profit-Ziel überschritten hat. Ist dies der Fall, werden *profit\_target\_price* sowie *stop\_loss\_price* neu gesetzt, um den möglichen Gewinn eines Kaufes zu maximieren (**Zeile 30-32**).

Als letztes werden innerhalb des Backtests implementierte Methoden zur Erzeugung eines HTML Dashboards aufgerufen und die im Backtest gesammelten Statistiken auf der Konsole ausgegeben (**Zeile 35-40**).

#### 5.4.2. Erstellung eines Dashboards

Unter der Verwendung der Bibliothek *plotly* ergibt sich die Erstellung eines Finanzdiagramms als kein großes Hindernis. Für die Erstellung eines Candlestick Charts gibt es speziell dafür vorgefertigte Klassen, denen die Daten überreicht werden müssen:

**binance.py**


---

```

1  def create_candlestick_figure(self):
2      df = self.candlestick_df
3
4      # Plot candlestick chart
5      candle = Candlestick(
6          x=df["time"],
7          open=df["open"],
8          close=df["close"],
9          high=df["high"],
10         low=df["low"],
11         name="Candlesticks"
12     )
13     data = [candle]
14
15     # Loop through all indicators of the market data and plot them
16     for indicator in self.strategy.indicators:
17         if type(indicator) == SimpleMovingAverage:
18             sma = Scatter(
19                 x=df["time"],
20                 y=df[indicator.name],
21                 name=indicator.name,
22                 line=dict(color="rgba(255, 207, 102, 1)")
23             )
24             data.append(sma)
25
26     # Customize style and display
27     layout = Layout(
28         xaxis={
29             "title": self.symbol,
30             "rangeslider": {"visible": True},
31             "type": "date"
32         },
33         yaxis={
34             "fixedrange": False,
35             "title": "Price per coin"
36         }
37     )
38
39     # Create figure and plot it
40     figure = Figure(data=data, layout=layout)
41     return figure

```

---

Quelltext 5.13: Erzeugung eines Candlestick Chart

Erklärungen zum Code:

**1. Erzeugen des Candles (Zeile 5-13):**

Mithilfe der *plotly* Library wird ein Candlestick-Objekt erzeugt, dem die jeweiligen Daten übergeben werden. Anschließend wird das Objekt in eine Liste eingefügt, welche alle Daten des Diagramms enthält.

**2. Erzeugen der Indikatoren (Zeile 16-24):**

Für jeden Indikator, den die gewählte Strategie besitzt, wird ein sogenannter Scatter-Plot erzeugt, mit welchem sich Linien und Punkte darstellen lassen. Ihm werden die jeweiligen Zeiten und Indikatorwerte übergeben.

**3. Festlegen eines Layouts (Zeile 27-41):**

Bevor die erstellten Objekte in eine *Figure* überführt werden können, welche die Objekte zu einem Diagramm vereint, kann ein Layout festgelegt werden. Mithilfe des Layouts können beispielsweise Titel und Achsenbeschriftungen angepasst werden.

Die Erstellung des Diagramms zur Visualisierung des Kapitals lehnt sich stark an die oben gezeigte Erstellung des Candlestick Charts an. Die Substitution der Statistikdaten in den vordefinierten HTML Code läuft jedoch anders ab.

Zunächst wird eine HTML Datei definiert, welche die Darstellung der Daten vorgibt. Mit dem nachfolgenden HTML Code kann eine Darstellung wie in Abbildung 5.3 erreicht werden:

**binance.py**


---

```

1  </p>
2  <hr />
3  <p>Symbol: {symbol_var}</span></p>
4  <p>API: {api_var}</span></p>
5  <p>Strategy: {strategy_var}</span></p>
6  <p>Trading fee: {trading_fee_var}%</span></p>
7  <p>Buy quantity: {buy_quantity_var} coins</span></p>
8  <p>Starting capital: {starting_capital_var}&euro;</span></p>
9  <p><br /></p>
10 <p>

```

---

Quelltext 5.14: HTML Vorlage

```

Symbol: {symbol_var}
API: {api_var}
Strategy: {strategy_var}
Trading fee: {trading_fee_var}%
Buy quantity: {buy_quantity_var} coins
Starting capital: {starting_capital_var}€

```

Abbildung 5.3.: Ergebnis eines Ausschnitts von HTML Code

Wie sich leicht erkennen lässt, sind alle in Mengenklammern enthaltenen Variablen wie *symbol\_var* Platzhalter für echte Werte. Diese müssen nun vom Backtest substituiert werden, was erreicht werden kann, indem innerhalb der Methode *stats\_to\_html()* gleichnamige Variablen mit den zugehörigen Werten initialisiert werden, welche dann die Plätze der HTML Variablen einnehmen.

#### backtest.py

---

```

1  def __stats_to_html(self):
2      symbol_var = self.symbol
3      api_var = self.api.base
4      strategy_var = self.strategy.name
5      trading_fee_var = self.api.trading_fee * 100
6      buy_quantity_var = self.buy_quantity
7      starting_capital_var = self.starting_capital
8      # ...
9
10     # Get html code from file
11     path = os.path.join(get_project_root(),
12                          "src/backtest/backtest_stats.html")
13     f = open(path, "r")
14     html_code = f.read()
15
16     # Substitute variables in html code with local variables from here
17     html_code = html_code.format(**locals())
18     return html_code

```

---

Quelltext 5.15: Substitution von HTML Variablen

Die Deklaration der gleichnamigen Variablen findet sich in Zeile 2-7 wieder. Anschließend muss der vordefinierte HTML Code, der sich noch in einer HTML Datei befindet, eingelesen werden (Zeile 11-13). Am Ende werden alle Platzhalter durch die richtigen Werte substituiert und der HTML Code an den Aufrufer zurückgegeben (Zeile 16-17).

Nach Verarbeitung aller zu visualisierenden Daten muss der HTML Code in eine Datei gespeichert werden, welche dann von herkömmlichen Browsern aufgerufen werden kann. Dazu wird für jeden Backtest automatisch eine Ordnerstruktur erstellt, falls sie nicht bereits besteht:

#### backtest.py

---

```

1  project_root = Path(__file__).parent.parent.parent
2  dashboard_dir = project_root + "dashboards/" + strategy.name.replace(" ",
3  "_") + "/" + symbol
4  Path(dashboard_dir).mkdir(parents=True, exists_ok=True)

```

---

Quelltext 5.16: Erzeugen einer Ordnerstruktur für Backtest Dashboards

Zunächst wird die Projektwurzel ermittelt (Zeile 1), dann wird der Pfad zur Ordnerstruktur festgelegt indem die zu erstellenden Ordnernamen zur Projektwurzel

hinzugefügt werden (Zeile 2). Abschließend erzeugt der Aufruf einer Methode der Klasse *Path* die gegebene Ordnerstruktur, falls sie nicht bereits besteht.

Wurde die Ordnerstruktur angelegt, wird der jeweilige HTML Code als HTML Datei abgespeichert. Der Name der Datei wird automatisch generiert und besteht aus dem verwendeten Trading Symbol sowie dem Datum des Backtests. Im Zusammenspiel kann somit eine automatisch generierte Ordnerstruktur inklusive Dashboards, wie in Abbildung 5.4, erzeugt werden.

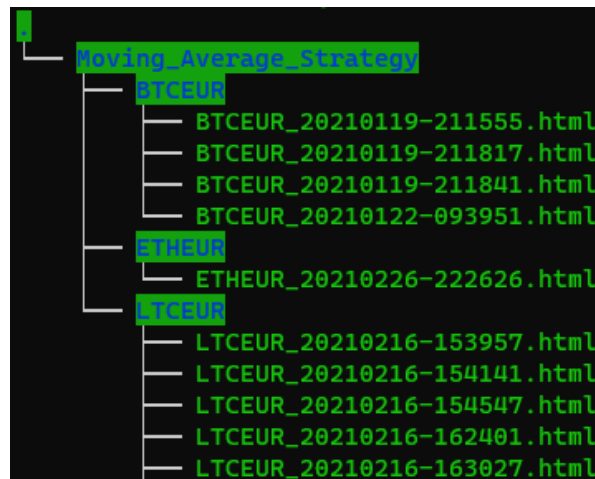


Abbildung 5.4.: Dashboard Ordnerstruktur

## 5.5. Trading Bot

Da der Trading Bot innerhalb dieser Arbeit keine echten Funktionalitäten bieten soll, fällt dessen Implementierung schwächer aus als bei manch anderen Komponenten. Ein grober Überblick über die enthaltenen Funktionalitäten und Abhängigkeiten lässt sich jedoch schon mit einem Blick auf die Klassendefinition erahnen:

### bot.py

---

```

1 class Bot:
2     STATUS_RUNNING = "running"
3     STATUS_INIT = "init"
4     STATUS_ABORTED = "aborted"
5     STATUS_PAUSED = "paused"
6
7     def __init__(self, name, symbol, api, strategy, starting_capital,
8         buy_quantity, description=""):
9         self.id = -1
10        self.name = name
  
```



```

10     self.symbol = symbol
11     self.api = api
12     self.strategy = strategy
13     self.starting_capital = starting_capital
14     self.capital = starting_capital
15     self.buy_quantity = buy_quantity
16     self.description = description
17     self.market_data = self.__get_init_data()
18     self.buy_signals = list()
19     self.buy_transactions = list()
20     self.sell_transactions = list()
21     self.not_sold_yet = dict()
22     self.status = self.STATUS_INIT

```

---

Quelltext 5.17: Klassendefinition des Trading Bots

Erklärung der Klassenvariablen:

- **STATUS\_RUNNING**: Bot ist aktiv
- **STATUS\_INIT**: Der Bot wurde erst erstellt bzw. initialisiert und noch nie verwendet.
- **STATUS\_ABORTED**: Der Bot wurde durch einen Laufzeitfehler gestoppt und wartet auf eine Fehlerbehebung bzw. den Neustart.
- **STATUS\_PAUSED**: Der Bot wurde durch den Nutzer pausiert.

Soll ein neuer Bot instanziiert werden, muss diesem Name, Trading Symbol, API, Strategie, Startkapital und Kaufmenge übergeben werden. Außerdem kann eine optionale Beschreibung hinzugefügt werden, welche beispielsweise Informationen über dessen Verwendung enthalten könnte.

### 5.5.1. Repräsentation der Marktdaten

Wurde ein neuer Bot instanziiert, muss er mit einem Datensatz von Marktdaten initialisiert werden, damit er mit dem Trading beginnen kann. Jeder Datensatz eines Trading Bots wird durch die Klasse *MarketData* repräsentiert:

#### market\_data.py

---

```

1  class MarketData:
2
3      def __init__(self, symbol, init_times, init_prices):
4          logger.info("Creating new market data...")
5          self.symbol = symbol
6          self.times = init_times
7          self.prices = init_prices

```

---

Quelltext 5.18: Repräsentation der Marktdaten

**Zeile 3-7:** Jedes Marktdatenobjekt besitzt eine Instanzvariable *symbol*, welche das Trading Symbol enthält, über welche es die Preisdaten verfügt. In den beiden Listen *times* und *prices* werden immer eine feste Menge von Preis und zugehörigen Zeitdaten gehalten.

Weitere, hier nicht dargestellte Funktionalitäten, sind das Erzeugen eines DataFrames aus seinen Marktdaten und die Möglichkeit den neuesten Preis inklusive zugehöriger Zeit abzufragen. Außerdem muss ein solcher Datensatz ständig aktualisiert werden, darf sich dabei jedoch nicht vergrößern. Daher werden beim Hinzufügen neuer Werte am Ende, die ältesten Werte am Anfang der Listen entfernt.

### 5.5.2. Beschaffen und Aktualisieren der Marktdaten

Die Initialisierung der Marktdaten geschieht in Verbindung mit der Schnittstelle *Binance* welche das Beschaffen der Daten übernimmt:

#### **bot.py**

---

```
1 def __get_init_data(self):
2     limit = 2 * 30 * 24 * 60
3     candlestick_df = self.api.get_candlestick_data(symbol=self.symbol,
4         interval="1M", limit=limit)
5
6     # Extract only the time and price columns
7     times = candlestick_df["time"].tolist()
8     prices = candlestick_df["close"].tolist()
9
10    # Create market data object
11    market_data = MarketData(self.symbol, times, prices)
12    return market_data
```

---

Quelltext 5.19: Baschaffung der Marktdaten

Anders als der Backtest, soll jeder Bot einen viel genaueren Datensatz von Preisdaten verwalten. Während beim Backtest ein Kerzenintervall von einer Stunde festgelegt wurde, ist es hier ein Zeitintervall von einer Minute. Im obigen Code-Ausschnitt in Zeile 4 wird ein Limit für einen Datensatz berechnet, welcher zwei Monate an minütigen Preisdaten enthalten soll.

Beim Aktualisieren der Daten fordert der Trading Bot per *Binance* Schnittstelle die Serverzeit der Plattform und den aktuellen Preis an, welcher dann zum *MarketData* Objekt hinzugefügt wird.

## 5.6. Bot Runner

Auch der Bot Runner erhält eine nur spärliche Implementierung, da dessen echte Funktionalität nicht gegeben sein soll. Stattdessen soll das Grundgerüst dessen definiert und in die Konsolenanwendung eingebaut werden. Ein Hinzufügen von Bots zum Bot Runner soll jedoch gegeben sein, um erstellte Trading Bots später in der Konsolenanwendung anzeigen lassen zu können:

### bot\_runner.py

---

```
1 class BotRunner:
2
3     def __init__(self):
4         self.bot_id = 0
5         self.bots = dict()
6
7     def add_bot(self, bot):
8         new_id = self.bot_id + 1
9         bot.id = new_id
10        self.bots[bot.id] = bot
11        self.bot_id = new_id
12        return new_id
```

---

Quelltext 5.20: Klassendefinition des Bot Runners

Bei Instanziierung des *BotRunner* erhält er zwei Instanzvariablen. *bot\_id* ist eine Zählervariable, welche den Bots ihre ID vergibt, während die andere Instanzvariable die Key/Value Datenstruktur der zu verwaltenden Bots darstellt. Dort wird jeder Bot mit seiner ID als Key und seiner Instanz als Wert gespeichert. Jedes Mal, wenn ein neuer Bot hinzugefügt wird, inkrementiert sich der Wert der *bot\_id*.

## 5.7. Konsolenanwendung

Mit der Implementierung der Konsolenanwendung vervollständigt sich das Trading System. Die Klasse *CommandLineInterface* bündelt alle anderen Komponenten und stellt den Einstiegspunkt der Anwendung dar. Sie initialisiert bei ihrer Instanziierung nur eine einzelne Instanzvariable - den *BotRunner*.

Jede nutzbare Funktion der Konsolenanwendung erhält seinen eigenen Header, welcher bei nahezu jeder Aktion oben ausgegeben wird. Für die mehrmalige Verwendung dieser Header, wird eine Datei *headers.py* erzeugt, welche sie beinhaltet. Ein Beispiel eines solchen Banners ist der Folgende, welcher im Hauptmenü der Anwendung gezeigt wird:

**headers.py**


---

```

1  tab_str = "      " # 4 blanks
2
3  HEADER_WELCOME = (
4      "#####\n"
5      "#" + tab_str * 2 + "Welcome!" + tab_str * 2 + "#\n"
6      "#####\n"
7  )

```

---

Quelltext 5.21: Definition der Header

**5.7.1. Implementierung der Optionsauswahl**

Wie bereits im Lösungskapitel (4.6) beschrieben, soll die Konsolenanwendung dem Interaktionsprinzip der Optionsauswahl folgen. Dabei wird dem Nutzer eine Frage gestellt, die er mit der Eingabe einer Zahl beantwortet. Da diese und viele andere nützliche Hilfsfunktionen wiederverwendet werden, wurde sie innerhalb einer Utility Datei namens *cli\_util.py* platziert.

**cli\_util.py**


---

```

1  def choose_option(title, options, header, note=None):
2      user_input = ""
3      while user_input == "":
4          display_header(header)
5          text = FormattedText([("class:bold", title)])
6          print_formatted_text(text, style=style)
7
8          # Display all options
9          for option in options:
10             print(option)
11             print("")
12             if note:
13                 text = FormattedText([("class:yellow", "Note: " + note)])
14                 print_formatted_text(text, style=style)
15                 print("")
16
17             # Get user input
18             user_input = prompt("Enter number: ", validator=NumberValidator())
19             return int(user_input)

```

---

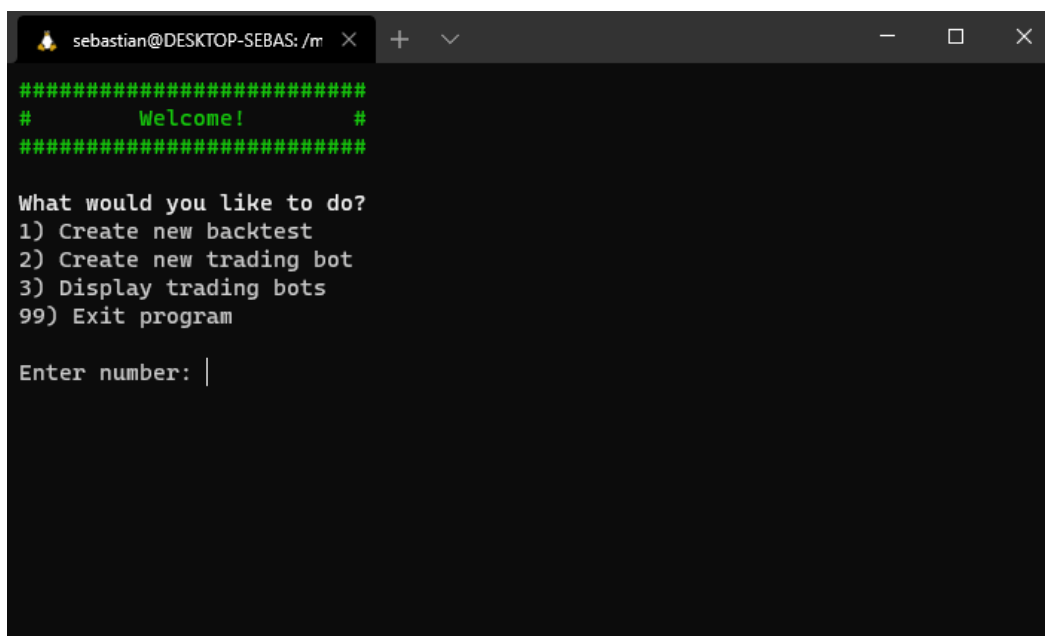
Quelltext 5.22: Implementierung der Optionsauswahl

Beim Aufruf der Methode müssen mehrere Übergabeparameter gegeben sein. Der *title* legt die Frage der Option fest - also beispielsweise "What would you like to do?". Der Parameter *options* besteht aus einer Liste, welche die verschiedenen, für den Nutzer auswählbaren, Optionen beinhaltet. Der *header* ist der jeweilige Header,

welcher angezeigt werden soll, während der letzte Parameter eine optionale Notiz beinhalten kann, der dann auf dem Terminal ausgegeben wird.

Zunächst werden Header und Optionstitel ausgegeben (**Zeile 4-6**). Beim Header geschieht dies über eine kleine Hilfsfunktion, welche gleichzeitig noch die Funktion hat, vorangegangene Ausgaben vom Konsolenfenster zu entfernen. Die formatierte Ausgabe des Optionstitel geschieht über eine Library für Konsolenanwendungen namens *prompt\_toolkit*. Anschließend werden alle übergebenen Optionen auf der Konsole ausgegeben. Wurde eine Notiz übergeben, wird auch diese noch angezeigt (**Zeile 12-15**).

Abschließend wird der User zur Eingabe einer Zahl gebeten, welche bestimmt, für welche Option er sich entschieden hat (**Zeile 18-19**). Dieser bisher beschriebene Vorgang wird solange wiederholt, bis der User eine richtige Eingabe tätigt (Schleife in **Zeile 3**). Durch einen Aufruf dieser Funktion mit den geeigneten Übergabeparametern wird eine Optionsauswahl, wie in Abbildung 5.5 dargestellt, erreicht.



```
sebastian@DESKTOP-SEBAS: /m
#####
#           Welcome!           #
#####

What would you like to do?
1) Create new backtest
2) Create new trading bot
3) Display trading bots
99) Exit program

Enter number: |
```

Abbildung 5.5.: Optionsauswahl des Main Menu

### 5.7.2. Validierung der Nutzereingaben

Damit der Nutzer dabei unterstützt wird, richtige Eingaben zu tätigen, wird eine Validierung der Nutzereingaben benötigt. Dies fördert sowohl die User Experience des Users und dient gleichzeitig dem Schutz vor Laufzeitfehlern durch falsche Eingabewerte. Mithilfe der bereits angesprochenen Library *prompt\_toolkit* kann

beim Aufruf der User-Input-Funktion *prompt()* ein solcher Validierer übergeben werden, der die Nutzereingabe in Echtzeit überprüft. Im nachfolgenden Listing befindet sich ein Zahlenvalidierer wie er bei der Optionsauswahl benutzt wird.

#### **validators.py**

---

```
1 class NumberValidator(Validator):
2     def validate(self, document):
3         text = document.text
4
5         if text and not text.isdigit():
6             index = 0
7             # Get index of first non numeric character
8             for index, char in enumerate(text):
9                 if not char.isdigit():
10                    break
11            raise ValidationError(message="This input contains non-numeric
                characters", cursor_position=index)
```

---

Quelltext 5.23: Implementierung zur Validierung von Nutzereingaben

Mithilfe des *prompt\_toolkit* kann eine Klasse *NumberValidator* erstellt werden, die von der Klasse *Validator* erbt. Jedes Mal, wenn der Nutzer eine Eingabe tätigt, springt die Eingabefunktion in den Zahlenvalidierer, um dessen Eingabe zu prüfen. Hat der Nutzer eine Eingabe getätigt, die nicht komplett aus Zahlen besteht (**Zeile 5**) wird die Eingabe mithilfe einer Schleife Zeichen für Zeichen überprüft, um die fehlerhafte Stelle zu finden (**Zeile 8-10**). Hat er die Stelle gefunden, wird ein *ValidationError* erzeugt, welcher den Cursor an die Stelle der fehlerhaften Eingabe platziert und eine Fehlernachricht ausgibt - dargestellt in Abbildung 5.6.

```

sebastian@DESKTOP-SEBAS: /m
#####
#           Welcome!          #
#####

What would you like to do?
1) Create new backtest
2) Create new trading bot
3) Display trading bots
99) Exit program

Enter number: f|

This input contains non-numeric characters

```

Abbildung 5.6.: Beispiel eines Validierungsfehlers bei fehlerhafter Eingabe

### 5.7.3. Erstellung von Backtests und Bots

Alle in Abbildung 5.5 dargestellten Funktionalitäten werden in der Hauptdatei *cli.py* untergebracht. Entscheidet sich der Nutzer beispielsweise dazu, einen neuen Backtest zu erstellen, muss er Schritt für Schritt durch verschiedene Auswahlmöglichkeiten begleitet werden. In diesen Auswahlmöglichkeiten entscheidet der Nutzer über die zu verwendende Konfiguration des Backtests:

#### **validators.py**

---

```

1  def create_backtest(self):
2      # Backtest config
3      api = choose_api(HEADER_NEW_BACKTEST)
4      if api == 99:
5          return
6      symbol = choose_symbol(HEADER_NEW_BACKTEST)
7      if symbol == 99:
8          return
9      # ...
10
11     # Check config and ask whether the user wants to start the backtest
12     display_header(HEADER_NEW_BACKTEST)
13     print_bold("Check configuration: ")
14     print("")
15     print(f"API: {api.base}")
16     print(f"Symbol: {symbol}")

```

```

17     print(f"Strategy: {strategy.name}")
18     print(f"Time frame: {time_frame_name}")
19     print(f"Starting capital: {starting_capital}")
20     print(f"Buy quantity: {buy_quantity}")
21     print("")
22     user_input: str = prompt("Do you want to start the backtest? [y/n] ",
                              validator=YesNoValidator())
23     print("")
24
25     if user_input == "y":
26         # Create backtest
27         backtest: Backtest = Backtest(symbol, api, strategy,
                                         starting_capital, buy_quantity, kline_limit)
28         backtest.run()
29         input("Press Enter to continue...")
30     elif user_input == "n":
31         return

```

---

Quelltext 5.24: Erstellung eines neuen Backtest

Zunächst durchläuft der Nutzer durch eine Auswahl an Möglichkeiten, in denen er Dinge wie die zu verwendende API, Trading Symbol, Strategie, etc. auswählen muss (**Zeile 3-8**). Hat er die Konfigurationsauswahl durchlaufen, wird die verwendete Konfiguration nochmal angezeigt und gefragt, ob der Nutzer den Backtest starten möchte (**Zeile 12-23**). Bestätigt der User die Frage mit einem *y*, wird eine neue Instanz des Backtests erzeugt und gestartet.

Soll ein neuer Trading Bot erstellt werden, wird dieses Prinzip in fast der gleichen Manier angewendet, weshalb sich die Erstellung des Backtest kaum von der Erstellung eines Trading Bots unterscheidet.

Der Erstellungsvorgang eines neuen Backtests, sowie Abbildungen zur Nutzung aller weiterer Funktionen befinden sich in Anhang A.



## 6. Evaluierung

Um den funktionellen und persönlichen Erfolg dieses Projektes messen zu können, muss zuvor definiert werden, auf welcher Grundlage diese Entscheidung getroffen werden soll. Eine gute Entscheidungsgrundlage stellt der Rückblick auf die im Projekt definierten Ziele dar.

Hauptziel der Arbeit war es, ein algorithmisches Handelssystem für Kryptowährungen zu entwerfen und zu implementieren. Dabei sollte es dem System möglich sein, Marktdaten zu sammeln und, mithilfe von entwickelten Strategien und Indikatoren durch einen Backtest, zu analysieren. Des weiteren sollte das Fundament zur Erstellung und Nutzung von Trading Bots konzipiert und implementiert werden. Eine weitere Aufgabe war das Erstellen einer Konsolenanwendung, welche die zuvor genannten Funktionalitäten vereint und dem Nutzer zur Verfügung stellt. Diese Ziele sollten im Rahmen eines Semesters erarbeitet und dokumentiert werden.

Ausgehend von einer funktionellen Betrachtung wurden diese Ziele und ihre Anforderungen in der gegebenen Zeit mit einem sehr zufriedenstellendem Ergebnis erreicht. Im Hinblick auf den Arbeitsaufwand, welcher zur Erreichung dieser Ziele betrieben wurde, müssen jedoch negative Rückschlüsse gezogen werden. Da es zu Beginn des Projektes sehr schwer war, den vollen Umfang der einzelnen Features zu ermitteln, gestaltete sich das Erreichen dieser Ziele, vor allem zum Ende hin, zunehmend schwieriger. So wurde der Dokumentationsaufwand, alle Komponenten ausreichend und lückenlos zu dokumentieren, deutlich unterschätzt. Leider wurde keine Funktion zur Messung der aufgebrauchten Zeit verwendet, jedoch wird davon ausgegangen, dass der zeitliche Umfang dieser Arbeit den geforderten zeitlichen Umfang deutlich überschreitet. Ebenso wurde der geforderte Dokumentationsaufwand überschritten, was ebenso auf die schwer zu treffende Einschätzung des Arbeitsaufwandes zurückzuführen ist - was jedoch aus Gründen der Vollständigkeit dann nicht mehr vermieden werden konnte.

Ein sehr positiver Aspekt dieser Arbeit, war die Anwendung vieler verschiedener Technologien und Konzepte, welche das Projekt sehr interessant und vielfältig gemacht haben. So konnten viele, im Studium erlernte, Themen und Verfahren angewandt und geübt werden. Zusätzlich wurde mit diesem System eine Grundlage für das Weiterarbeiten mit der Thematik *Algorithmic Trading* geschaffen, was vor allem ein persönliches Ziel darstellte.

# 7. Zusammenfassung und Ausblick

## 7.1. Erreichte Ergebnisse

Zusammenfassend lässt sich sagen, dass so gut wie alle, für das Projekt gesteckte Ziele erreicht wurden. So konnte innerhalb eines Semesters folgender Fortschritt erzielt werden:

- Entwurf eines Architekturkonzepts für das Trading System
- Entwurf und Implementierung einer Schnittstelle zur Kommunikation einer REST API
- Eine geeignete Verarbeitung der von der API gelieferten Daten
- Entwurf eines Konzepts zur modularen Einbindung von Trading Strategien und deren Indikatoren
- Testen der Strategien durch einen geeigneten Backtest
- Visualisierung und Speicherung der Backtest-Ergebnisse
- Konzeption und Entwurf eines Trading Bots für die zukünftige Verwendung des Trading Systems
- Konzeption und Entwurf einer geeigneten Verwaltung dieser Trading Bots
- Vereinen aller Features in einer konsistenten und intuitiven Konsolenanwendung

## 7.2. Ausblick

Wie bereits erwähnt, soll das in dieser Arbeit entwickelte Projekt den Grundstein einer Weiterentwicklung bilden. Daher gibt es beispielsweise eine Vielzahl an Ideen für Features und weitere Komponenten, welche im Rahmen dieser Arbeit nicht umgesetzt werden konnten.

**Erweiterung des Systems um eine Datenbank**

Ohne das Persistieren von Trading Bot Konfigurationen und getätigten Transaktionen, ist ein Betrieb im echten Umfeld nicht sinnvoll. Vom System getätigte Informationen müssen automatisch gespeichert und abgerufen werden können. Dazu kann ein geeignetes Datenbankkonzept entworfen und Implementiert werden.

### **Implementierung echter Funktionalitäten für reales Handeln**

Um eine Verwendung im echten Umfeld möglich zu machen, müssen die noch benötigten Funktionalitäten zum Trading Bot hinzugefügt werden. Hierfür müssen verschiedene Order Types und Trading Konzepte umgesetzt werden.

### **Automatisches Testen aller Systemkomponenten**

Läuft das System einmal im realen Umfeld, muss natürlich gewährleistet werden, dass alle Systemkomponenten fehlerfrei funktionieren. Hier bietet sich beispielsweise die Verwendung eines Testing Frameworks an, welches in Verbindung mit Unit- oder Integrationstests eine automatische Überprüfung der Komponenten durchführen kann.

### **Aufsetzen eines Webservers für das Handelssystem**

Damit das System in der Lage ist, Tag und Nacht handeln zu können, ist das Aufsetzen eines Webservers dafür durchaus sinnvoll. Erreicht werden kann dies z.B. mit einem RaspberryPi und einem Apache Webserver.

### **Erstellung einer Webapplikation zur Überwachung und Verwaltung des Handelssystems**

In weiter Zukunft, wenn das System alle wichtigen Funktionalitäten abdeckt, bietet es sich an, eine Webapplikation für das Handelssystem zu entwerfen. So könnte beispielsweise ein Dashboard aufgerufen werden, welches Echtzeitergebnisse des Systems darstellt und visualisiert. Zusätzlich könnte so eine Verwaltung und Steuerung der Trading Bots bzw. des gesamten Systems geschaffen werden.

### **Erweiterung des Backtests**

Auch der Backtest ist noch nicht perfekt. Hier muss vor allem die Verwendung von Gleitkommazahlen durch die Verwendung der Python Klasse *Decimal* ersetzt werden, um eine perfekte Berechnungsgenauigkeit zu erzielen. Außerdem könnte der Backtest so erweitert werden, dass alle Strategien auf alle Coins getestet werden können, wodurch sich die erfolgreichsten Strategien automatisch ermitteln lassen würde.

# Literatur

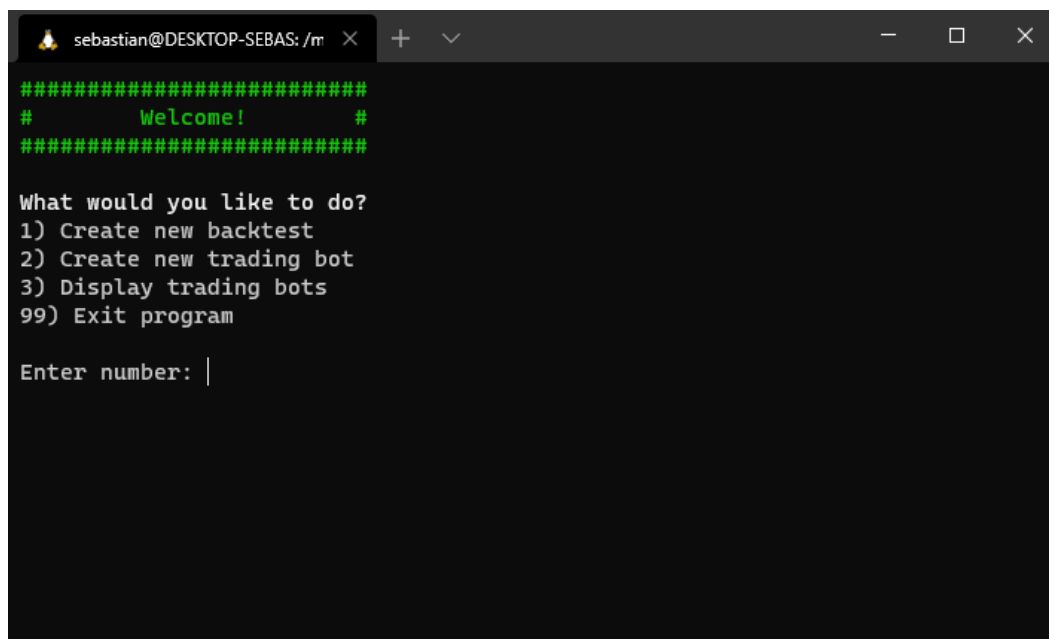
- [1] admiralmarkets.com. *So lesen und verstehen Sie ein Candlestick Chart*. 28. Mai 2020. URL: <https://admiralmarkets.com/de/wissen/articles/forex-basics/alles-was-sie-uber-candlesticks-wissen-mussen>.
- [2] Apexcharts. *Candlestick Chart*. 18. Feb. 2021. URL: <https://apexcharts.com/vue-chart-demos/candlestick-charts/basic/>.
- [3] Blocktrainer. *Was ist Mining?* 18. Aug. 2020. URL: <https://www.youtube.com/watch?v=4w71joFRyJg>.
- [4] James Chen. *Trading Strategy*. 30. Apr. 2019. URL: <https://www.investopedia.com/terms/t/trading-strategy.asp#:~:text=A%20trading%20strategy%20is%20a,used%20to%20make%20trading%20decisions..>
- [5] coin-update.de. *Bitcoin Kurs 2011*. 3. Feb. 2021. URL: <https://coin-update.de/bitcoin-kurs/>.
- [6] Tammy Da Costa. *Moving Average (MA) Explained for Traders*. 13. Aug. 2019. URL: <https://www.dailyfx.com/education/technical-analysis-tools/moving-average.html>.
- [7] Daniel Dob. *Binance Review: The World's Leading Cryptocurrency Exchange?* 4. Feb. 2021. URL: <https://blockonomi.com/binance-review/>.
- [8] FinanceScout24. *Chancen und Risiken von Trading: Ein Einsteiger-Guide*. 17. Feb. 2021. URL: <https://www.financescout24.de/wissen/ratgeber/trading-einsteiger-guide>.
- [9] finanzen.net. *Bitcoin-Rekordjagd geht ungebremsst weiter*. 17. Feb. 2021. URL: <https://www.finanzen.net/nachricht/devisen/neues-allzeithoch-bitcoin-rekordjagd-geht-ungebremsst-weiter-9820255>.
- [10] Finanzfluss. *Bitcoins Erklärung: In nur 12 Min. Bitcoin verstehen!* 29. Aug. 2017. URL: <https://www.youtube.com/watch?v=2473NHJtdFA>.
- [11] Jean Folger. *The Basics of Trading a Stock: Know Your Orders*. 28. Jan. 2021. URL: <https://www.investopedia.com/investing/basics-trading-stock-know-your-orders/>.
- [12] Jake Frankenfield. *Cryptocurrency*. 5. Mai 2020. URL: <https://www.investopedia.com/terms/c/cryptocurrency.asp>.
- [13] geeksforgeeks. *Python | Pandas DataFrame*. 10. Jan. 2019. URL: <https://www.geeksforgeeks.org/python-pandas-dataframe/>.

- [14] Ronald Gehrt. *Das Super-Tool Candlestick-Charts: Was Sie wissen müssen*. URL: <https://www.lynxbroker.de/boerse/trading/technische-analyse/charttypen/das-super-tool-candlestick-charts-was-sie-wissen-muessen/>.
- [15] Beate Krol. *Kreislauf des Geldes*. 22. Feb. 2021. URL: <https://www.planet-wissen.de>.
- [16] Ben Lobel. *Technical Indicators Defined and Explained*. 24. Juni 2019. URL: <https://www.dailyfx.com/education/technical-analysis-tools/technical-indicators.html>.
- [17] Sagar Mane. *Understanding REST*. 9. Juni 2017. URL: <https://medium.com/@sagar.mane006/understanding-rest-representational-state-transfer-85256b9424aa>.
- [18] MuleSoft. *What is an API?* 22. Feb. 2021. URL: <https://www.mulesoft.com/resources/api/what-is-an-api>.
- [19] pypi.org. *Project description*. 9. Feb. 2021. URL: <https://pypi.org/project/pandas/>.
- [20] python.org. *What is Python? Executive Summary*. 22. Feb. 2021. URL: <https://www.python.org/doc/essays/blurb/>.
- [21] Kai Schiller. *Bitcoin Trading 2020*. 11. Aug. 2020. URL: <https://blockchainwelt.de/bitcoin-trading-alle-infos-zum-handel/>.
- [22] Shobhit Seth. *Basics of Algorithmic Trading: Concepts and Examples*. 7. März 2020. URL: <https://www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp>.
- [23] spiegel.de. *Bitcoin knackt Marke von 50.000 Dollar*. 16. Feb. 2021. URL: <https://www.spiegel.de/wirtschaft/unternehmen/bitcoin-knackt-marke-von-50-000-dollar-a-5b2ddc78-2a45-49d2-a38e-d580f940675e>.
- [24] André Stagge. *Trading: Was ist das und welche Handelsarten gibt es?* 27. Okt. 2019. URL: <https://www.nextmarkets.com/de/handel/glossar/trading>.
- [25] Rheinwerk Verlag. *Grundlegende Konzepte*. 22. Feb. 2021. URL: [http://openbook.rheinwerk-verlag.de/python/02\\_001.html](http://openbook.rheinwerk-verlag.de/python/02_001.html).
- [26] Dr. Carsten Weerth. *Fundamentaldaten*. 19. Feb. 2021. URL: <https://wirtschaftslexikon.gabler.de/definition/fundamentaldaten-34165>.
- [27] Tim Weingaerter. *Blockchain und Bitcoin - die Buzzwords der Stunde*. 22. Feb. 2021. URL: <https://hub.hslu.ch/informatik/blockchain-einfach-erklaert/>.
- [28] Wikipedia. *Backtesting*. 23. Feb. 2021. URL: <https://de.wikipedia.org/wiki/Backtesting>.

- [29] Wikipedia. *Blockchain*. 22. Feb. 2021. URL: <https://en.wikipedia.org/wiki/Blockchain>.
- [30] Wikipedia. *Hypertext Markup Language*. 28. Feb. 2021. URL: [https://de.wikipedia.org/wiki/Hypertext\\_Markup\\_Language](https://de.wikipedia.org/wiki/Hypertext_Markup_Language).
- [31] Wikipedia. *Hypertext Transfer Protocol*. 22. Feb. 2021. URL: [https://de.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol).
- [32] Wikipedia. *Programmierschnittstelle*. 22. Feb. 2021. URL: <https://de.wikipedia.org/wiki/Programmierschnittstelle>.
- [33] Wikipedia. *Representational State Transfer*. 22. Feb. 2021. URL: [https://de.wikipedia.org/wiki/Representational\\_State\\_Transfer](https://de.wikipedia.org/wiki/Representational_State_Transfer).
- [34] youknow. *Blockchain in 3 Minuten erklärt*. 5. Sep. 2018. URL: <https://www.youtube.com/watch?v=4FU3tc-foaI>.

# A. Anhang zur Konsolenanwendung

## A.1. Main Menu



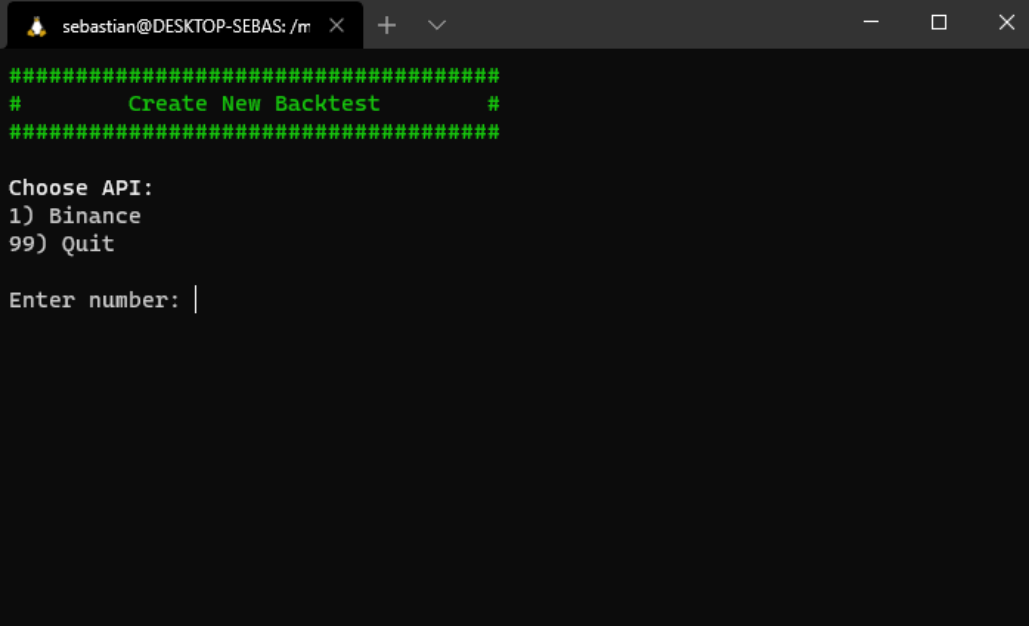
```
sebastian@DESKTOP-SEBAS: /m
#####
#           Welcome!           #
#####

What would you like to do?
1) Create new backtest
2) Create new trading bot
3) Display trading bots
99) Exit program

Enter number: |
```

Abbildung A.1.: Auswählen der API

## A.2. Erstellung eines Backtests

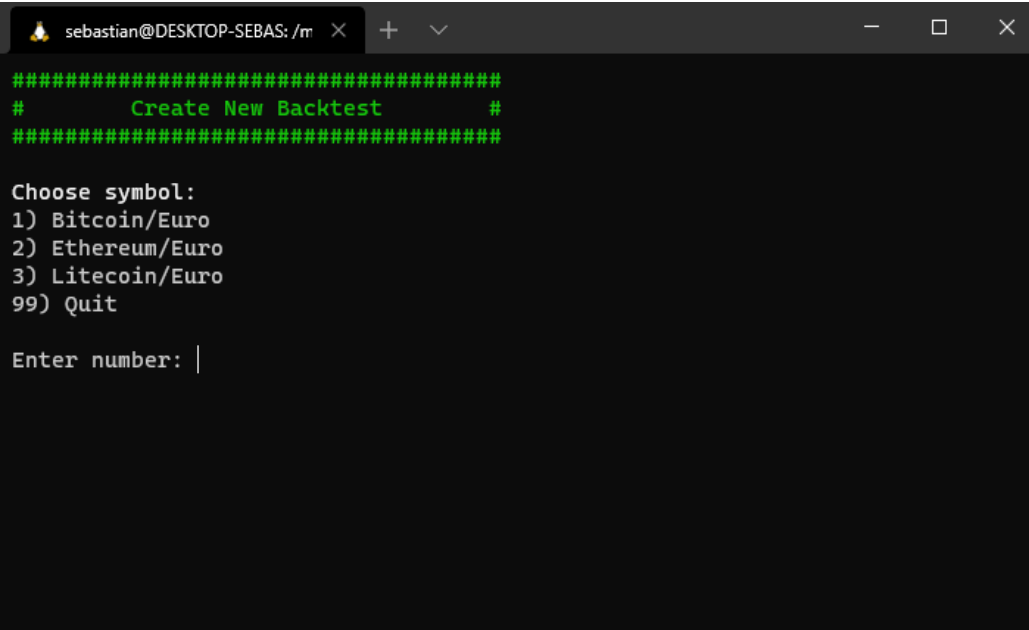


```
sebastian@DESKTOP-SEBAS: /m x + v
#####
#       Create New Backtest       #
#####

Choose API:
1) Binance
99) Quit

Enter number: |
```

Abbildung A.2.: Auswählen der API



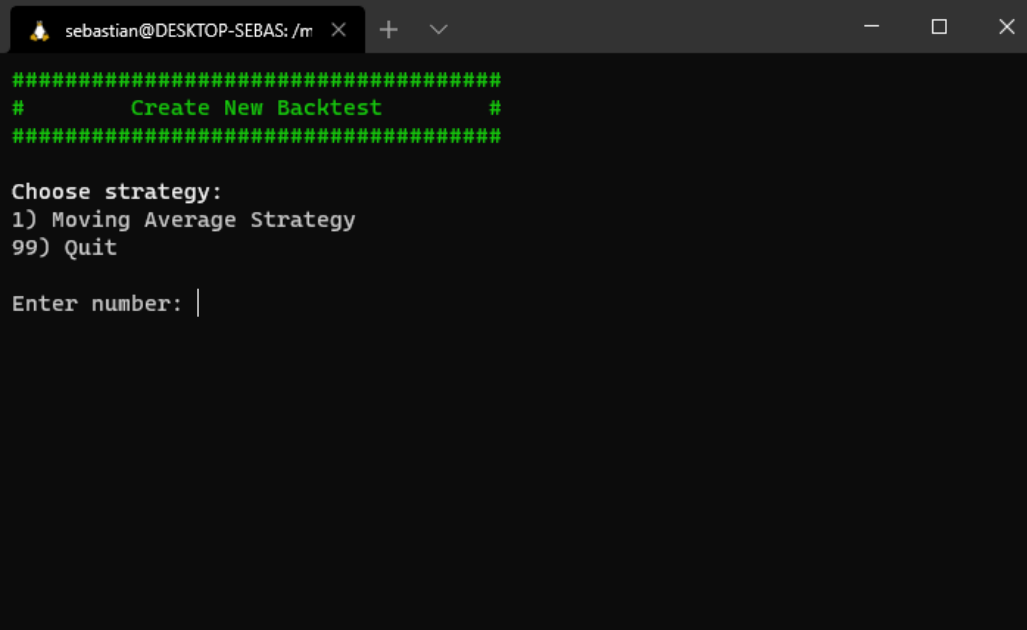
```
sebastian@DESKTOP-SEBAS: /m x + v
#####
#       Create New Backtest       #
#####

Choose symbol:
1) Bitcoin/Euro
2) Ethereum/Euro
3) Litecoin/Euro
99) Quit

Enter number: |
```

Abbildung A.3.: Auswählen des Trading Symbols



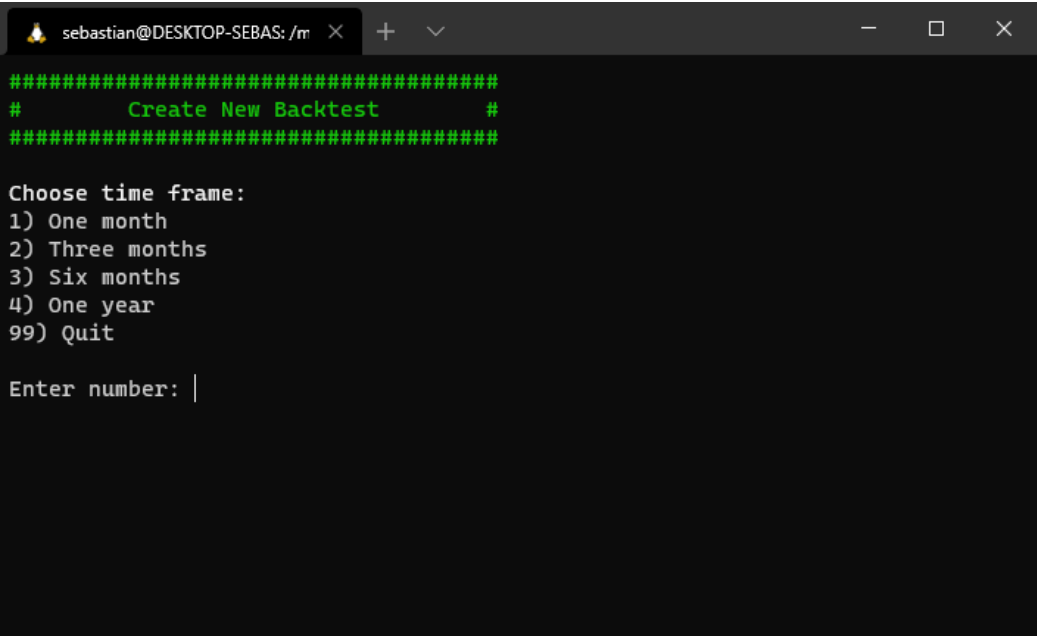


```
sebastian@DESKTOP-SEBAS: /m
#####
#       Create New Backtest       #
#####

Choose strategy:
1) Moving Average Strategy
99) Quit

Enter number: |
```

Abbildung A.4.: Auswählen der Strategie

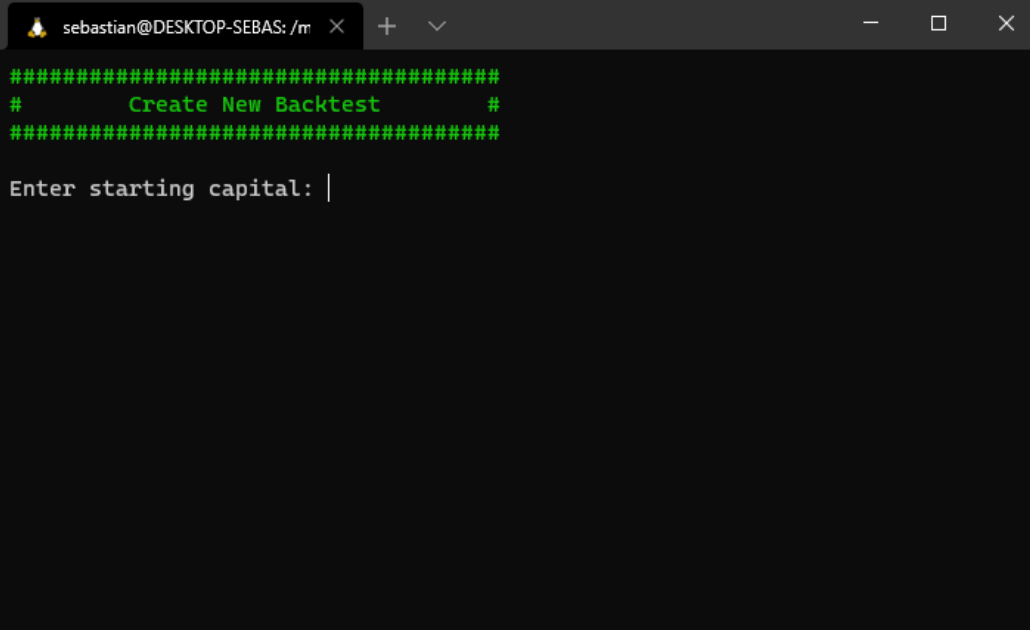


```
sebastian@DESKTOP-SEBAS: /m
#####
#       Create New Backtest       #
#####

Choose time frame:
1) One month
2) Three months
3) Six months
4) One year
99) Quit

Enter number: |
```

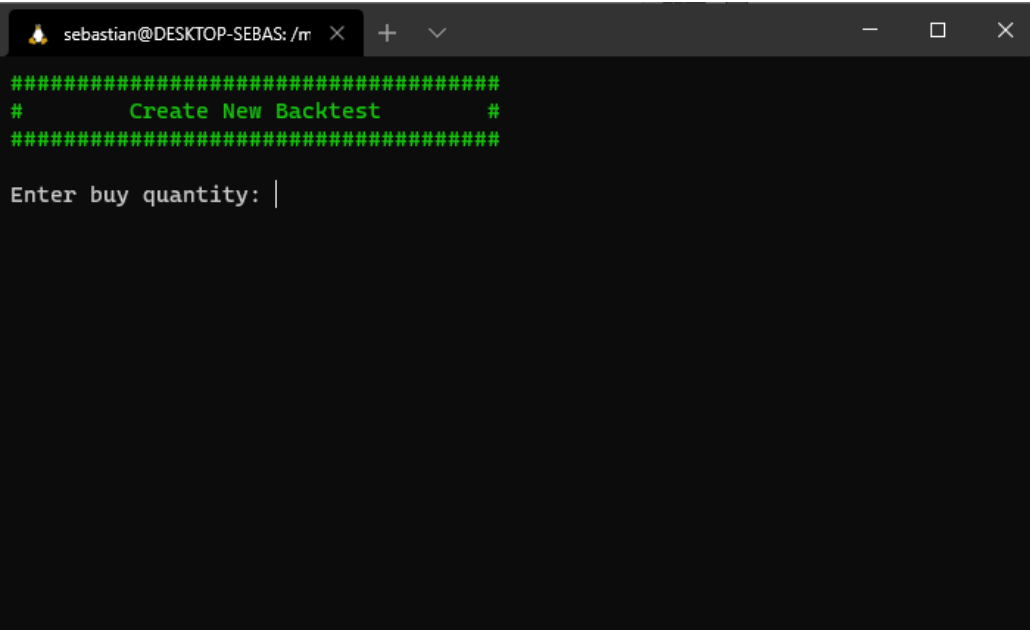
Abbildung A.5.: Auswählen der Dauer des Backtests



```
sebastian@DESKTOP-SEBAS: /m
#####
#       Create New Backtest       #
#####

Enter starting capital: |
```

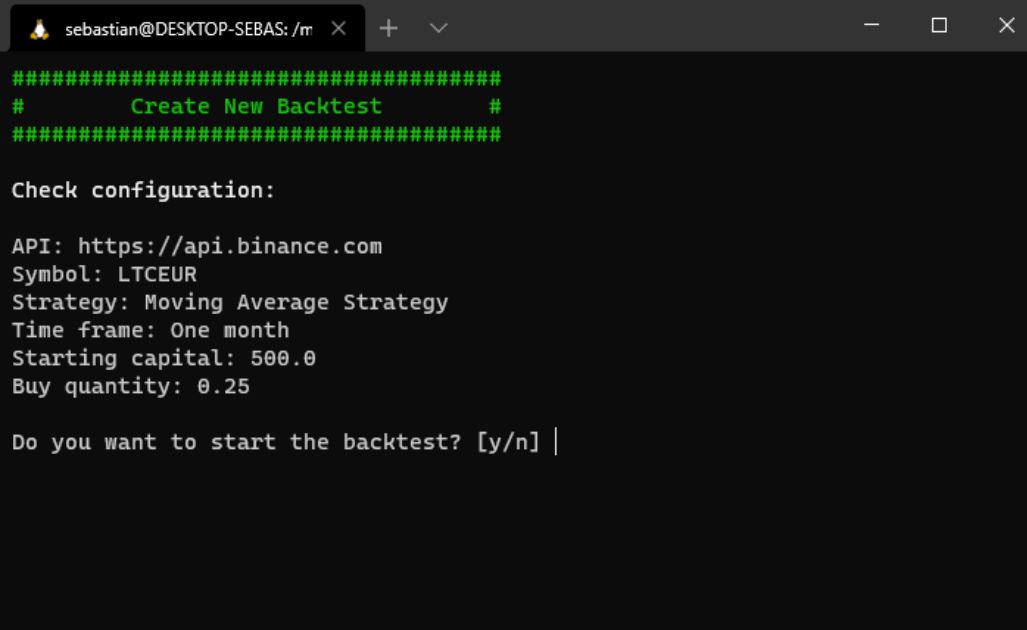
Abbildung A.6.: Eingabe des Startkapitals



```
sebastian@DESKTOP-SEBAS: /m
#####
#       Create New Backtest       #
#####

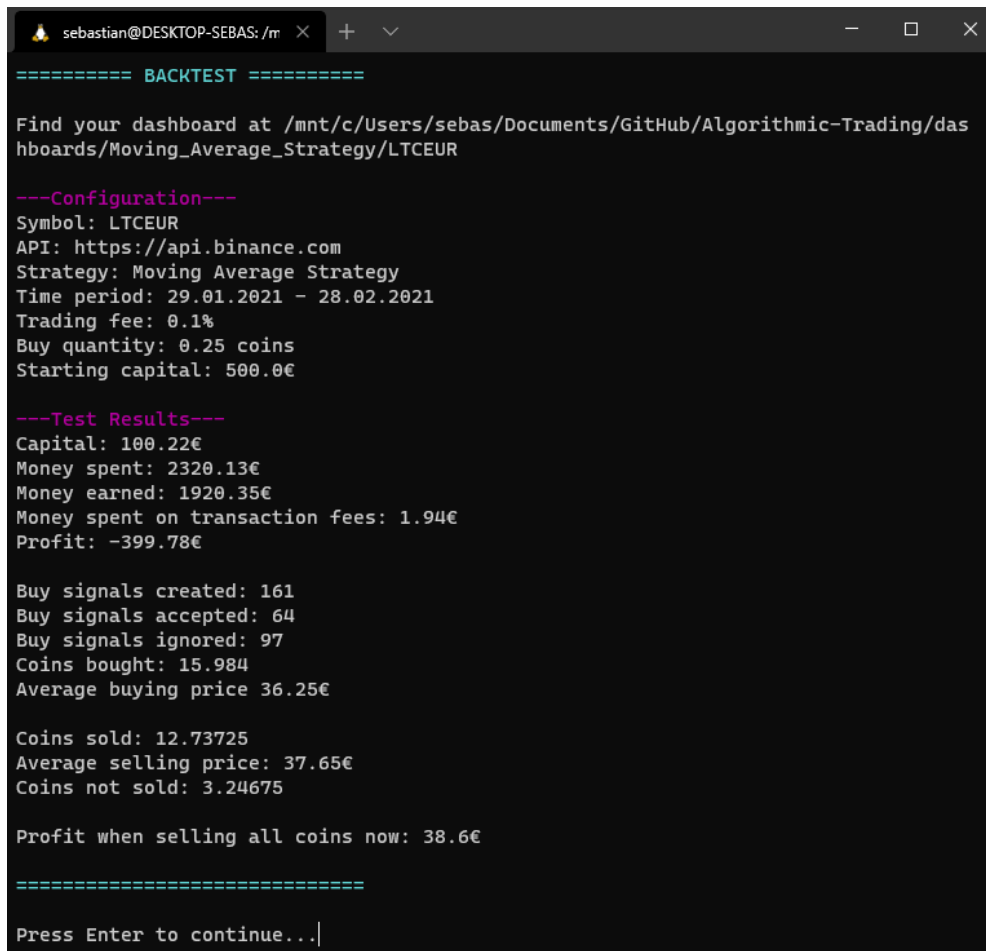
Enter buy quantity: |
```

Abbildung A.7.: Eingabe der Kaufmenge pro Kauf



```
sebastian@DESKTOP-SEBAS: /m × + ∨  
#####  
#           Create New Backtest           #  
#####  
  
Check configuration:  
  
API: https://api.binance.com  
Symbol: LTCEUR  
Strategy: Moving Average Strategy  
Time frame: One month  
Starting capital: 500.0  
Buy quantity: 0.25  
  
Do you want to start the backtest? [y/n] |
```

Abbildung A.8.: Ausgabe der Konfiguration

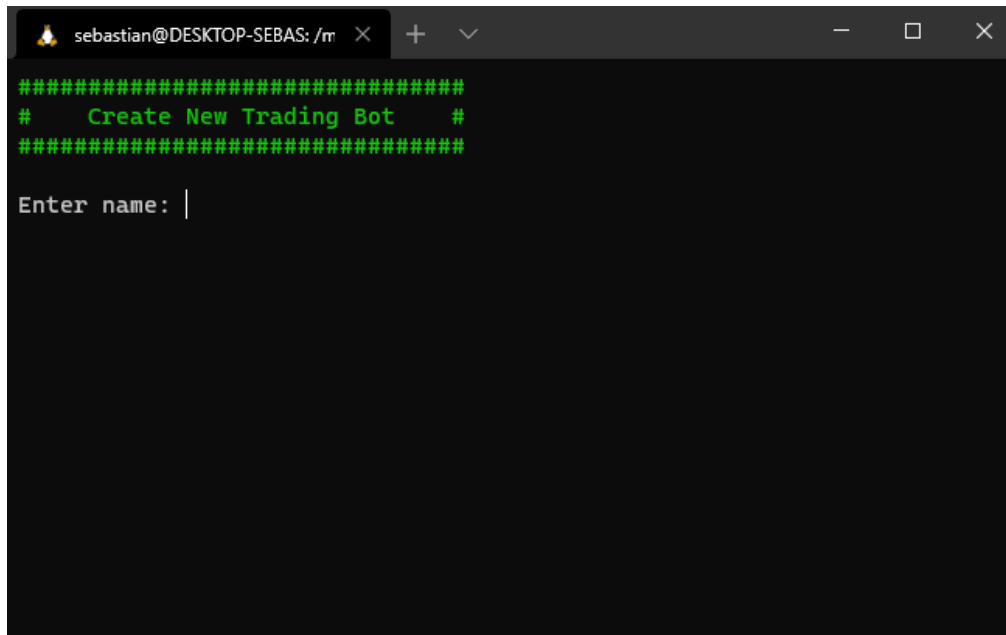


```
sebastian@DESKTOP-SEBAS: /m  × + ∨  
===== BACKTEST =====  
  
Find your dashboard at /mnt/c/Users/sebas/Documents/GitHub/Algorithmic-Trading/dashboards/Moving_Average_Strategy/LTCEUR  
  
---Configuration---  
Symbol: LTCEUR  
API: https://api.binance.com  
Strategy: Moving Average Strategy  
Time period: 29.01.2021 - 28.02.2021  
Trading fee: 0.1%  
Buy quantity: 0.25 coins  
Starting capital: 500.0€  
  
---Test Results---  
Capital: 100.22€  
Money spent: 2320.13€  
Money earned: 1920.35€  
Money spent on transaction fees: 1.94€  
Profit: -399.78€  
  
Buy signals created: 161  
Buy signals accepted: 64  
Buy signals ignored: 97  
Coins bought: 15.984  
Average buying price 36.25€  
  
Coins sold: 12.73725  
Average selling price: 37.65€  
Coins not sold: 3.24675  
  
Profit when selling all coins now: 38.6€  
  
=====
```

Press Enter to continue...|

Abbildung A.9.: Ausgabe der Backtest-Statistik

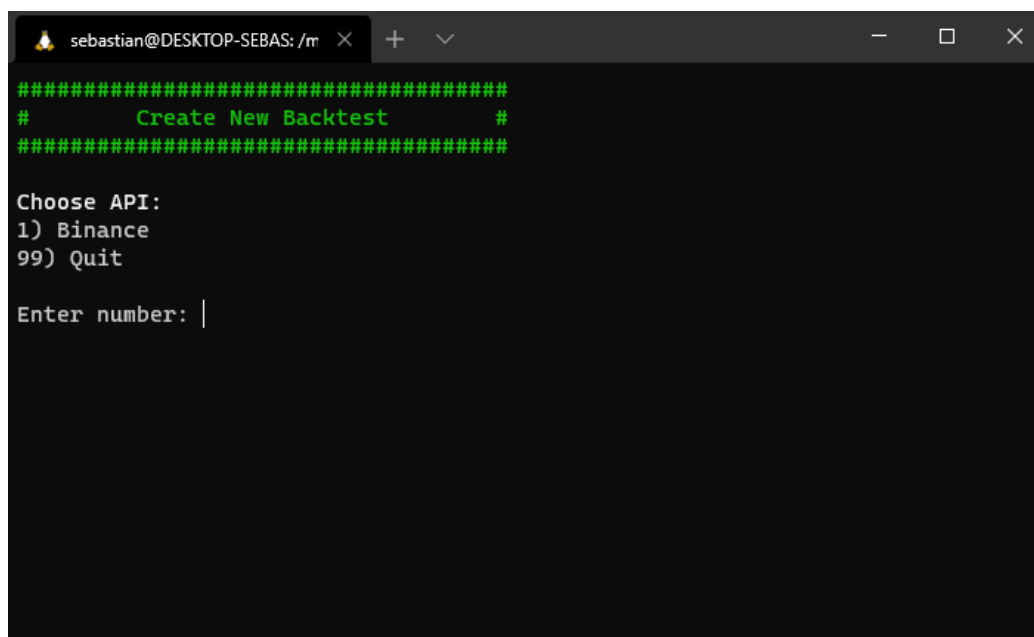
### A.3. Erstellung eines Trading Bots



```
sebastian@DESKTOP-SEBAS: /m x + v - □ x
#####
#   Create New Trading Bot   #
#####

Enter name: |
```

Abbildung A.10.: Auswählen der API

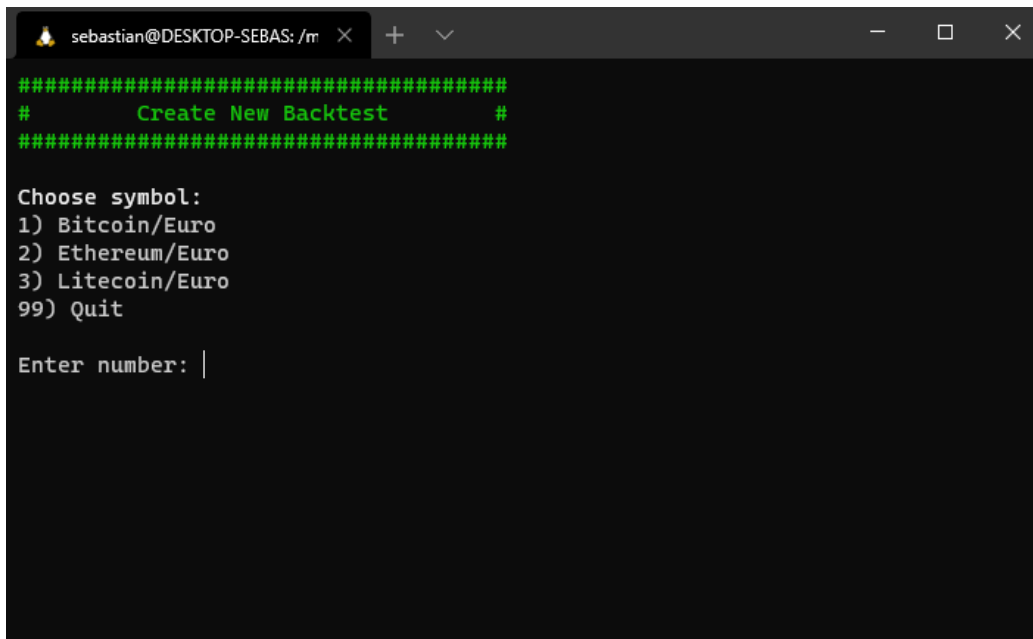


```
sebastian@DESKTOP-SEBAS: /m x + v - □ x
#####
#   Create New Backtest     #
#####

Choose API:
1) Binance
99) Quit

Enter number: |
```

Abbildung A.11.: Auswählen der API

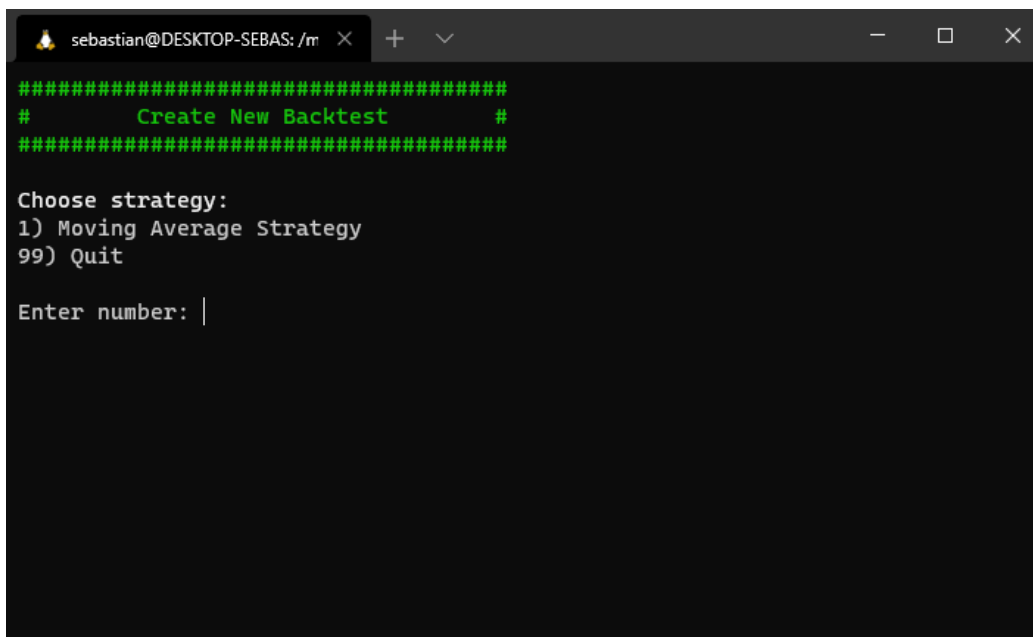


```
sebastian@DESKTOP-SEBAS: /m
#####
#      Create New Backtest      #
#####

Choose symbol:
1) Bitcoin/Euro
2) Ethereum/Euro
3) Litecoin/Euro
99) Quit

Enter number: |
```

Abbildung A.12.: Auswählen des Trading Symbols

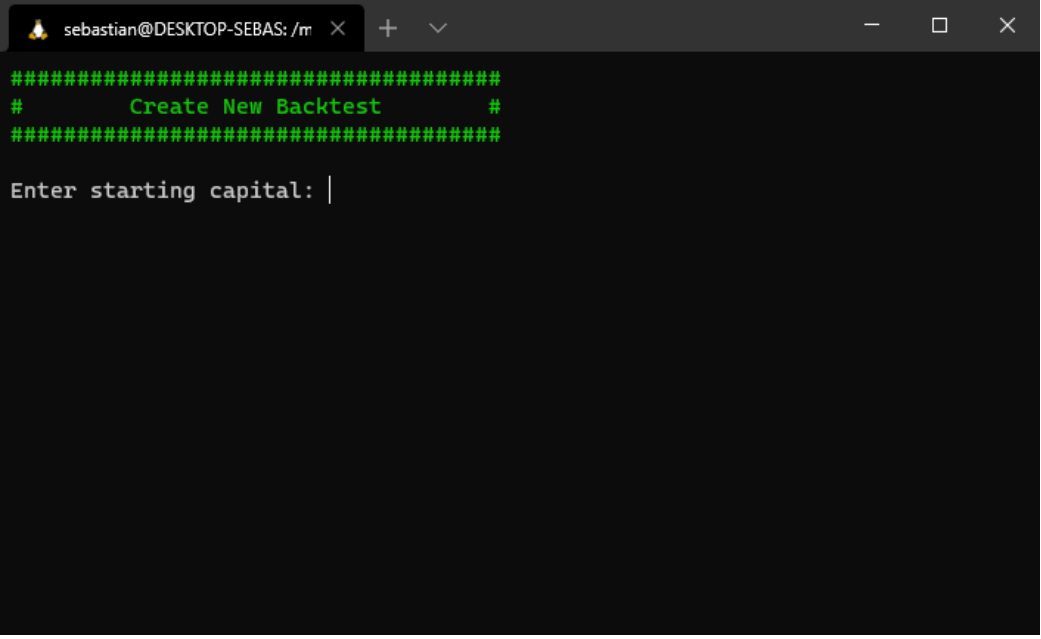


```
sebastian@DESKTOP-SEBAS: /m
#####
#      Create New Backtest      #
#####

Choose strategy:
1) Moving Average Strategy
99) Quit

Enter number: |
```

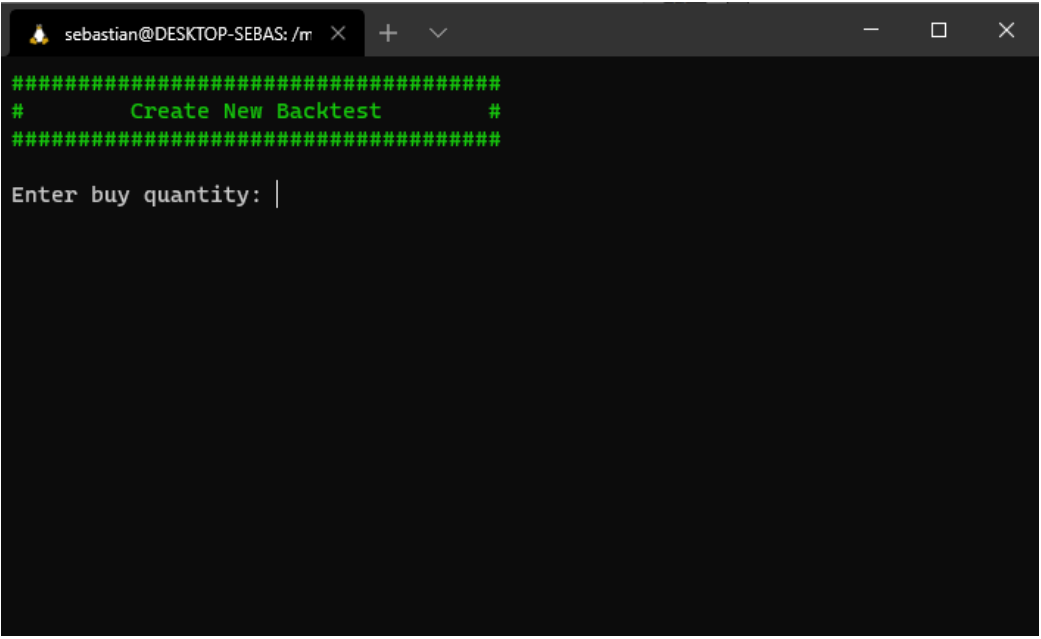
Abbildung A.13.: Auswählen der Strategie



```
sebastian@DESKTOP-SEBAS: /m X + v
#####
#      Create New Backtest      #
#####

Enter starting capital: |
```

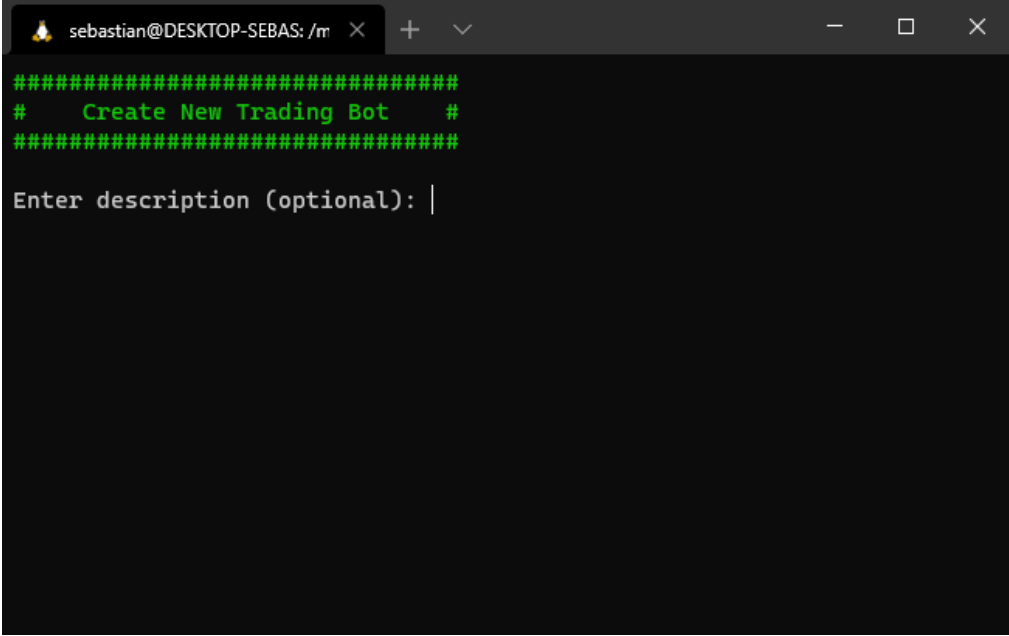
Abbildung A.14.: Eingabe des Startkapitals



```
sebastian@DESKTOP-SEBAS: /m X + v
#####
#      Create New Backtest      #
#####

Enter buy quantity: |
```

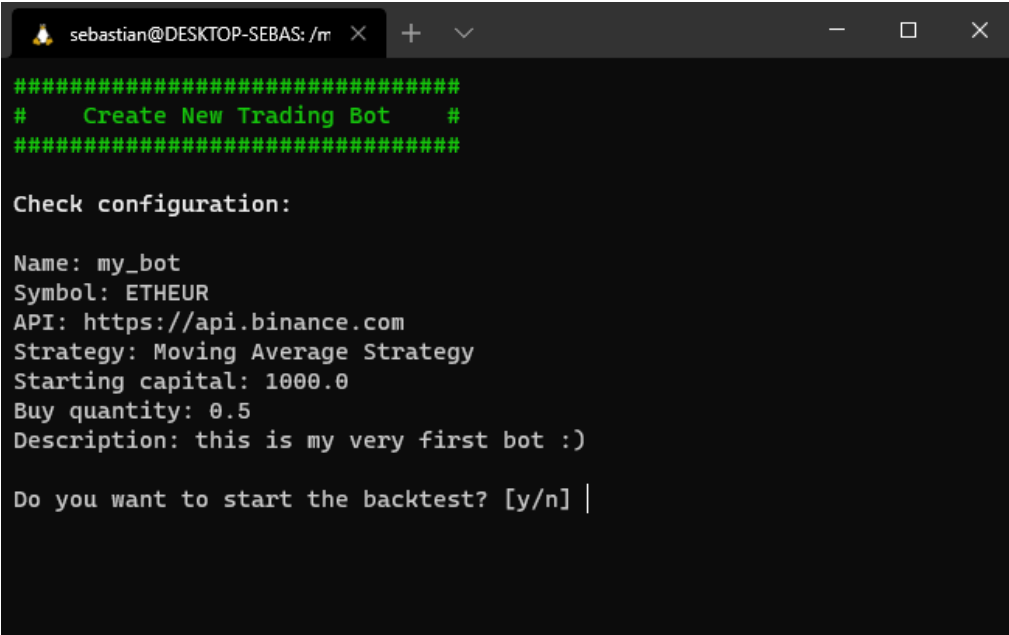
Abbildung A.15.: Eingabe der Kaufmenge pro Kauf



```
sebastian@DESKTOP-SEBAS: /m
#####
#   Create New Trading Bot   #
#####

Enter description (optional): |
```

Abbildung A.16.: Optimale Eingabe einer Beschreibung



```
sebastian@DESKTOP-SEBAS: /m
#####
#   Create New Trading Bot   #
#####

Check configuration:

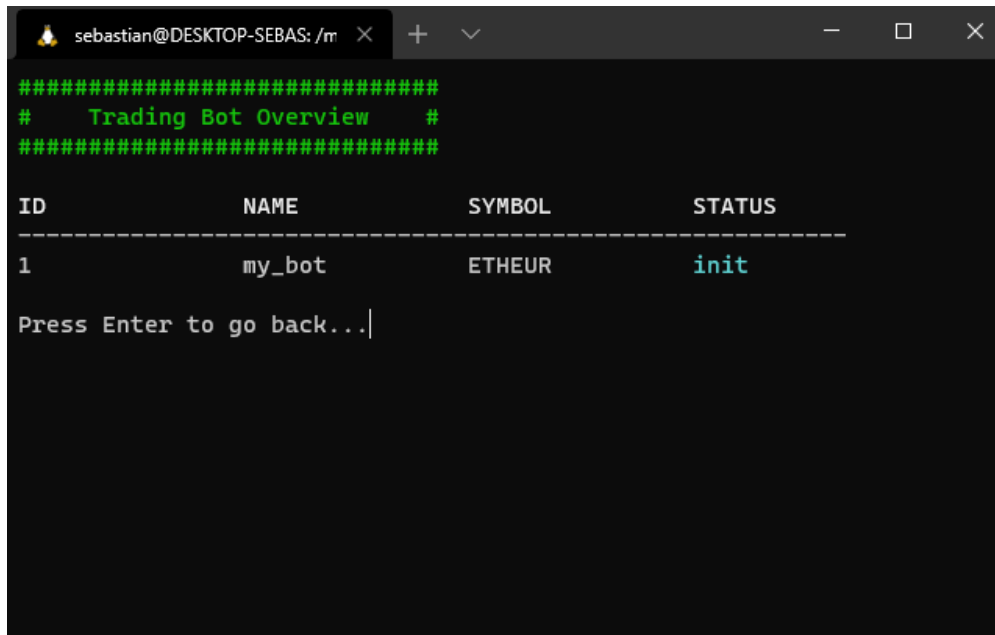
Name: my_bot
Symbol: ETHEUR
API: https://api.binance.com
Strategy: Moving Average Strategy
Starting capital: 1000.0
Buy quantity: 0.5
Description: this is my very first bot :)

Do you want to start the backtest? [y/n] |
```

Abbildung A.17.: Überprüfen der Konfiguration



## A.4. Anzeigen der erstellten Trading Bots

A terminal window titled 'sebastian@DESKTOP-SEBAS: /m' displays a 'Trading Bot Overview'. The overview is enclosed in a green border and shows a table with one bot. The bot has ID 1, name 'my\_bot', symbol 'ETHEUR', and status 'init'. Below the table, it says 'Press Enter to go back...|'.

```
#####  
#   Trading Bot Overview   #  
#####  
  
ID      NAME      SYMBOL      STATUS  
-----  
1       my_bot    ETHEUR     init  
  
Press Enter to go back...|
```

Abbildung A.18.: Übersicht der Erstellten Trading Bots