# Anwendungen der Mathematik in der Informatik

## Darstellung von unterschiedlichen ganz-rationalen Funktionen in Python

Jon Defilla

28. April 2019

# Inhaltsverzeichnis

# Abbildungsverzeichnis



# Tabellenverzeichnis

# 1 Einleitung

## 1.1 Motivation

Programmieren und Informatik waren seit jeher meine Lieblingsfächer. Aus diesem Grund habe ich mich für diese Facharbeit entschieden. Ich habe mich schon sehr früh gefragt, wie Spiele und Programme gemacht werden. Vor ein paar Jahren begann ich in Python zu programmieren und erstellte sehr einfache Programme mithilfe der Turtle-Bibliothek. Diese Bibliothek soll Kindern das Programmieren beibringen. Ich habe sicherlich viel daraus gelernt, ohne diese Bibliothek hätte ich inzwischen wahrscheinlich das Interesse an der Programmierung verloren. Diese Facharbeit ist eine Gelegenheit für mich, um meine Kenntnisse in Informatik allgemein und in der Programmierung mathematischer Algorithmen zu vertiefen.

## 1.2 Projekt Vorstellung

In diesem Projekt werde ich versuchen, die Programmierung auf mathematische Probleme anzuwenden. Die Idee ist, ein Programm zu erstellen, das dieselbe Mathematik anwendet, die ich im Gymnasium gelernt habe, ohne dabei fertige Funktionsbibliotheken dritter Parteien zu verwenden. Das Programm entspricht im Wesentlichen dem Back-End[1] eines Grafik-Taschenrechners. Die Annahmekriterien[2] für dieses Projekt sind:

- Polynomaddition und -subtraktion
- Polynomableitung
  - Extrempunkt-Berechnung
  - Tangenten-Berechnung
- Polynomintegral- und Flächeninhalt-Berechnung

Für dieses Projekt werde ich Python Version 3.6.6 benutzen.

## 1.3 Warum Python?

Python ist eine Skriptsprache, die für schnelles Prototyping[3] verwendet werden kann. Dadurch kann sich der Programmierer auf die Entwicklung des Algorithmus konzentrieren, anstatt sich beispielsweise mit der Speicherverwaltung zu beschäftigen. Es ist eine Programmiersprache mit einer der gesprochenen Sprache ähnlichen und somit leicht zu erlernenden Syntax. Sie ist weit entwickelt und ermöglicht auch komplexe Programmierungen. Die Hauptanwendungen von Python sind Webentwicklung, hauptsächlich Back-End, Hacking und wissenschaftliches Programmieren,

[1] Vorgänge, die für den Benutzer nicht sichtbar sind.

[2] Die Annahmekriterien sind äußerst nützlich, da sie Grenzen definieren, als Grundlage für Tests dienen und eine genaue Planung und Schätzung ermöglichen, wie aus „Clear Acceptance Criteria and Why They're Important“, RubyGarage, 05. März 2019, `https://rubygarage.org/blog/clear-acceptance-criteria-and-why-its-important` hervorgeht.

[3] Ein Prototyp ist eine frühe Probe, ein Modell oder eine Freigabe eines Produkts, das zum Testen eines Konzepts oder Prozesses entwickelt wurde, nach „Prototype“, Wikipedia, 10. Apr 2019, `https://en.wikipedia.org/wiki/Prototype`

einschließlich maschinellen Lernens, künstlicher Intelligenz, Datenanalyse, Numerik, Modellierung und Statistik. Dies macht diese Sprache zu einer perfekten Wahl für dieses Projekt.

Python ist eine interpretierte Sprache, das bedeutet, wenn ein Python-Code ausgeführt wird, muss er zur Laufzeit zuerst von einem Interpreter übersetzt werden, damit die CPU die Anweisungen verarbeiten kann. Dies geschieht zu Lasten der Leistung, der Vorteil ist jedoch die saubere Syntax, berichtet „What's the difference between compiled and interpreted language?", Stackoverflow, 23. Feb 2019, `https://stackoverflow.com/questions/2657268/whats-the-difference-between-compiled-and-interpreted-language`

# 2 Theoretische Vorüberlegungen

## 2.1 Polynomadditionen und- subtraktionen

Polynome sind Funktionen, die aus der Summe der Vielfachen der Potenzen einer Variablen bestehen. Sie haben also die Form $y = a_n\, x^n + a_{n-1}\, x^{n-1} + ... + a_1\, x + a_0$. Die Polynomaddition und -subtraktion sind einfach zu programmieren: Man addiert oder subtrahiert einfach jeden Koeffizienten $a_i$ mit dem entsprechenden Koeffizienten $b_i$ der anderen Funktion.

## 2.2 Berechnung der Extrempunkte

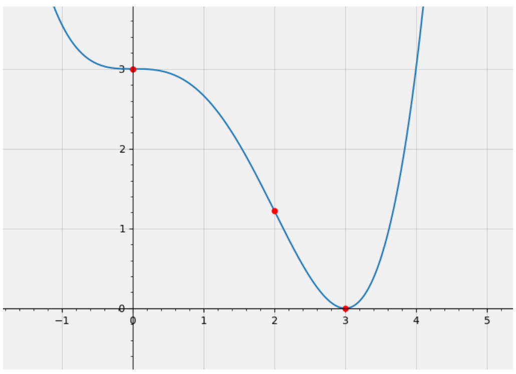

Abbildung 1: Funktion mit kritischen Punkten

Die Berechnung der kritischen Punkte einer Funktion, wie in Abbildung 1 mit roten Punkten dargestellt, erfordert Ableitungen sowie das Auffinden der Nullstellen einer Funktion. Die Verwendung der ersten Ableitung einer Funktion ermöglicht es, ihre Extrema zu finden. Diese befinden sich dort, wo die erste Ableitung verschwindet, also gilt $f'(x_e) = 0$. Die Lösung dieser Gleichung ergibt die Tief- und Hochpunkte, jedoch ist es möglich, die Punkte mit der zweiten Ableitung der Funktion zu unterscheiden. Die Tiefpunkte sind diejenigen, bei denen die zweite Ableitung positiv ist, also $f''(x_{min}) > 0$, die Hochpunkte sind diejenigen, bei denen die zweite Ableitung negativ ist, also $f''(x_{max}) < 0$ und die Sattelpunke diejenigen, bei denen die zweite Ableitung Null ist, also $f''(x_{sattel}) = 0$.

Die Ableitung einer Funktion entspricht ihrer Steigung. Eine Tangente ist eine Gerade, bzw. lineare Funktion, die eine Kurve an einem Punkt berührt und dort die gleiche Steigung hat. Sie hat die Funktion $y = mx + h$, wobei $m$ die Steigung der Geraden beschreibt und $h$ den Schnittpunkt mit der y-Achse. Es ist somit auch möglich, die Tangenten-Gleichung wie folgt zu schreiben: $y = f'(x) \cdot x + h$, wobei $x$ die freie Koordinate ist und $y$ das Bild von $x$. Die letzte fehlende Variable $h$ muss noch gefunden werden, um die vollständige Gleichung der Funktion zu erhalten.

Das Berechnen der Fläche zwischen einer Funktion und der x-Achse erfordert das Integral dieser Funktion. Der erste Schritt besteht darin, die Stammfunktion $F(x)$ der Funktion zu bestimmen. In einem zweiten Schritt kann das Integral dann in seinem x-Bereich beschränkt werden, indem $F(b) - F(a)$ gerechnet wird, wobei $b$ die obere Grenze und $a$ die untere Grenze des Integrals ist. Bei Kurven mit negativen Funktionswerten, wenn also ein Teil der Fläche unterhalb der x-Achse liegt, nimmt das Integral einen negativen Wert an. Es ist also erforderlich, den absoluten Wert der Funktion zu betrachten, bevor man die Fläche berechnet, wie im unten gezeigten Beispiel dargestellt ist:

$$\int_a^b |x^2 - 10|\,dx = \left[\left|\frac{1}{3}x^3 - 10x + c\right|\right]_a^b = \left|\frac{1}{3} \cdot b^3 - 10 \cdot b\right| - \left(\left|\frac{1}{3} \cdot a^3 - 10 \cdot b\right|\right)$$

## 2.3 Bestimmung der Nullstellen

Um die Nullstellen einer Funktion zu finden, habe ich mich für das Newton-Verfahren entschieden, weil es exponentiell schnell ist - man benötigt nur etwa 15 Iterationen, um die zwölfte Nachkommastelle einer Dezimalzahl zu finden. Obwohl die Newton-Methode viele Vorteile hat, kann sie nur verwendet werden, um eine Nullstelle einer Funktion zu finden, berichtet „Newton-Verfahren“, Wikipedia, 08. Jan 2019, `https://de.wikipedia.org/wiki/Newton-Verfahren`

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

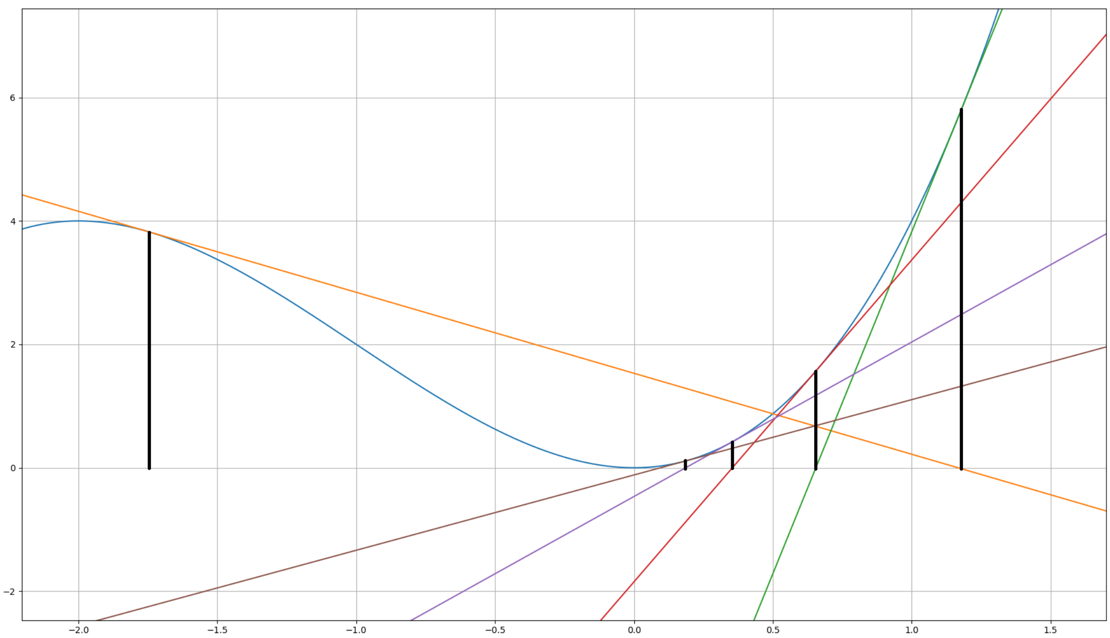


Abbildung 2: Newton-Verfahren

In Abbildung 2 ist in blau die Funktion $x^3 + 3x^2$ zusammen mit den von dem Newton-Verfahren erzeugten Tangenten dargestellt. Die Anfangskoordinate $x_0$ ist $-1,75$, bei der die Tangente die orange Linie ist. Diese orange Linie hat dann eine

Nullstelle bei 1.166, wo eine weitere Tangente gezogen wird. Diesen Prozess wiederholt sich, bis die gewünschte Genauigkeit erreicht ist, jedoch reichen bereits fünfzehn Iterationen aus.

Das Horner-Schema beschreibt eine Art der Polynomdivision. Die Funktion wird dabei mehrfach durch ein Polynom ersten Grades geteilt, um alle Nullstellen zu finden. Die Polynome haben dabei die Form $p = x - x_{NS}$, wobei $x_{NS}$ der x-Wert einer der Nullstellen ist. Es ist möglich, die Horner-Division in Verbindung mit der Newton-Methode zu verwenden, um einen vollständigen Algorithmus zu erstellen, der alle Nullstellen einer Funktion findet. Mithilfe der Newton Methode, wird eine erste Nullstelle gefunden. Im Rahmen des Horner-Schemas wird die Ausgangsfunktion dann durch das Polynom $p = x - x_{NS,Newton1}$ geteilt. Auf die dabei entstandene Funktion wird dann wieder das Newton-Verfahren angewendet um die nächste Nullstelle zu erhalten, die wiederum im Horner-Schema genutzt werden kann. Dieser Vorgang wird wiederholt, bis die Division gleich 1 ist. Mit Hilfe der Tabelle 1 und Abbildung 3 wird dieses Konzept visualisiert. Die Tabelle 1 zeigt den Prozess des Auffindens aller Nullen der Funktion $f(x) = x^3 + 2x^2 - x^2 - x - 2$.

Tabelle 1: Funktionsteilung

| Funktion | Nullstellen | Gefundene Nullstelle (Newton) | Nächste Division |
|---|---|---|---|
| $x^3 + 2x^2 - x - 2$ | -2.0; -1.0; 1.0 | 1.0 | $\frac{x^3+2x^2-x-2}{x-1}$ |
| $x^2 + 3x + 2$ | -2.0; -1.0 | $-1.0$ | $\frac{x^2+3x+2}{x-(-1)}$ |
| $x + 2$ | -2.0 | $-2.0$ | $\frac{x+2}{x-(-2)}$ |
| 1 | - | - | - |

# 3 Klassenmodell

Mein Programm verwendet eine Mischung aus funktionaler Programmierung[4] und objektorientierter Programmierung. Es ist in zwei Hauptteile gegliedert. Meine Polynomklasse (Back-End) und die Hauptfunktion, mit der das Menü meines Programms (Front-End) erstellt wird.

OOP[5] beinhaltet das Konzept von Objekten, die miteinander interagieren können. Diese Interaktion ist sehr hilfreich bei der Erstellung eines Datentyps, da jede Instanz der Klasse in meinem Fall unterschiedliche Koeffizienten haben kann und es sehr einfach ist, beispielsweise zwei Polynome zusammenzufügen, sagt „Objektorientierte Programmierung“, Wikipedia, 24. Feb 2019, `https://de.wikipedia.org/wiki/Objektorientierte_Programmierung`

Für die Menüfunktion habe ich mich für die funktionale Programmierung entschieden, obwohl ich auch OOP hätte verwenden können, weil es a) nur ein Menü

[4]Programmierung bei der Funktionen definiert werden.
[5]Object Oriented Programming

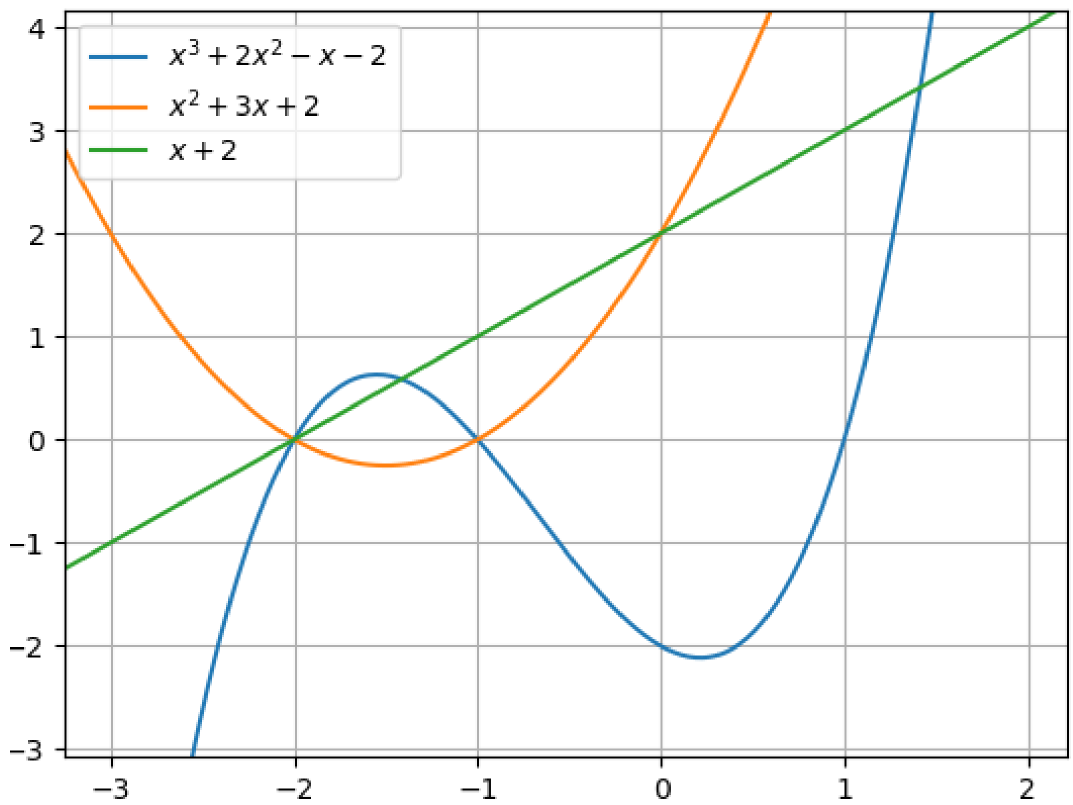


Abbildung 3: Funktion Division mit Newtons-Verfahren

gibt, so dass es nicht erforderlich ist, dass Instanzen mit sich selbst interagieren können und b) weil mein Menü im Wesentlichen eine unendliche Schleife ist, so dass OOP für diese Anwendung nicht sinnvoll ist.

Die Verwendung von OOP erfordert bestimmte Funktionen, die beim Einrichten des Objekts und bei der Definition von Operationen helfen. In Python werden diese Funktionen als „magische Funktionen“ bezeichnet, die von zwei Unterstrichen umgeben sind. Diese Funktionen werden implizit aufgerufen, wenn spezielle Operationen mit den Instanzen durchgeführt werden, z.B. wird die Funktion `__add__()` automatisch aufgerufen, wenn zwei Instanzen zusammengefügt werden, nach „Dunder or magic methods in Python“, GeeksForGeeks, 10. März 2019, `https://www.geeksforgeeks.org/dunder-magic-methods-python/`

# 4 Darstellung des Datenmodells

Es gibt ein Modell mit dem Namen „EVA Modell“, das die grundlegende Rolle einer Funktion oder eines Programms beschreibt: Eingabe, Verarbeitung und Ausgabe. Ein Programm nimmt eine Eingabe, z.B. eine Eingabe von einem Menschen oder einem Sensor, führt einige Berechnungen durch (Verarbeitung) und gibt sie dann in Form einer Nachricht an den Benutzer zurück oder speichert sie beispielsweise in einer Datei (Ausgabe). In den folgenden Kapiteln werde ich jeden Teil des EVA-Modells in meinem Programm darstellen.

## 4.1 Eingabe

### 4.1.1 Konstruktor

Ein Konstruktor ist immer die erste Funktion, die ausgeführt wird, wenn eine Instanz dieses Objekts erstellt wird. Daher wird er Konstruktor oder in Python die `__init__()` Methode genannt, was für Initialisierung steht. Diese „magische“ Methode definiert, wie eine Instanz erstellt werden sollte. Konventionell sollten alle Attribute der Klasse im Konstruktor benannt werden. Dies ist jedoch auch mit anderen Methoden möglich.

```
12 class Polynomial:
13     def __init__(self, coeffs: tuple):
14         if type(coeffs) != tuple:
15             raise TypeError("Only tuples are allowed.")
16
17         self.coeffs = coeffs
```

Fast jede Methode der Klasse benötigt einen `self` -Parameter, da die Methoden und das Attribut zur Instanz gehören. Wenn man eine Funktion in der Klasse aufrufen will, würde man zum Beispiel Folgendes eingeben:

```
instanz.calculate_area(4, 6)
```

Im Hintergrund ändert Python dies jedoch zum Folgenden:

```
Polynomial.calculate_area(instanz, 4, 6)
```

Aus diesem Grund ist das Schlüsselwort `self` immer erforderlich, außer bei `@staticmethod` und `@classmethod`. Das Schlüsselwort `self` ermöglicht die Verwendung von mehreren Instanzen mit völlig unterschiedlichen Werten. Der Parameter `coeffs` ist ein Tupel, das alle Koeffizienten der zu erstellenden Instanz enthält. Um Fehler zu vermeiden, habe ich eine Bedingung implementiert (Z. 14), die überprüft, ob `coeffs` ein Tupel ist, und einen Fehler auslöst (Z. 15), wenn dies nicht der Fall ist. Als Nächstes erstelle ich ein Attribut `coeffs` für diese Klasse und weise den als Argument übergebenen Wert zu (Z. 17). Wenn eine Instanz der Klasse erstellt wird, dürfen sich ihre Koeffizienten nicht ändern. Um Fehler zu vermeiden, habe ich die Koeffizienten in einem Tupel gespeichert, da ein Tupel im Gegensatz zu Arrays unveränderlich ist, also nicht versehentlich geändert werden kann.

### 4.1.2 Overloading

Nach der Initialisierung des Datentyps ist es sehr umständlich, eine Instanz zu erstellen, da der Benutzer alle Koeffizienten, einschließlich der Nullen, eingeben muss. Das heißt, wenn er z.B. die Funktion $x^2$ eingeben möchte, entspricht das der Eingabe (1, 0, 0) und nicht direkt der Funktion selbst. Deshalb wollte ich eine Funktion entwickeln, die diesen Prozess automatisiert, um es benutzerfreundlicher zu machen.

Der Begriff „Overloading“ wird in statisch typisierten Sprachen[6] verwendet, wenn man mehrere Funktionen mit demselben Namen erstellt, die einen etwas

[6] In einer statisch typisierten Sprache muss der Typ der Variablen explizit angegeben werden, nach „What is the difference between statically typed and dynamically typed languages?“, Stackoverflow, 05. Feb 2019, `https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages`

anderen Zweck haben. Diese Funktionen unterscheiden sich von ihren Parametern. Die entsprechende Funktion wird abhängig vom Typ und der Anzahl der vom Aufrufer übergebenen Argumente aufgerufen, wie es in „Computer Programming/Function overloading“, Wikipedia, 16. Dec 2018, `https://en.wikibooks.org/wiki/Computer_Programming/Function_overloading` heißt. Da Python eine dynamisch typisierte Sprache[7] ist, wird das „Overloading“ nicht unterstützt. Es gibt jedoch eine Alternative, die als `@classmethod` bezeichnet wird. Damit ist es möglich, „Overloading“ selbst zu erstellen, wie aus „Python classmethod“, Programiz, 07. Dec 2019, `https://www.programiz.com/python-programming/methods/built-in/classmethod` hervorgeht. In meinem Fall hat diese Funktion jedoch nicht denselben Namen, sondern fungiert als zweiter Konstruktor.

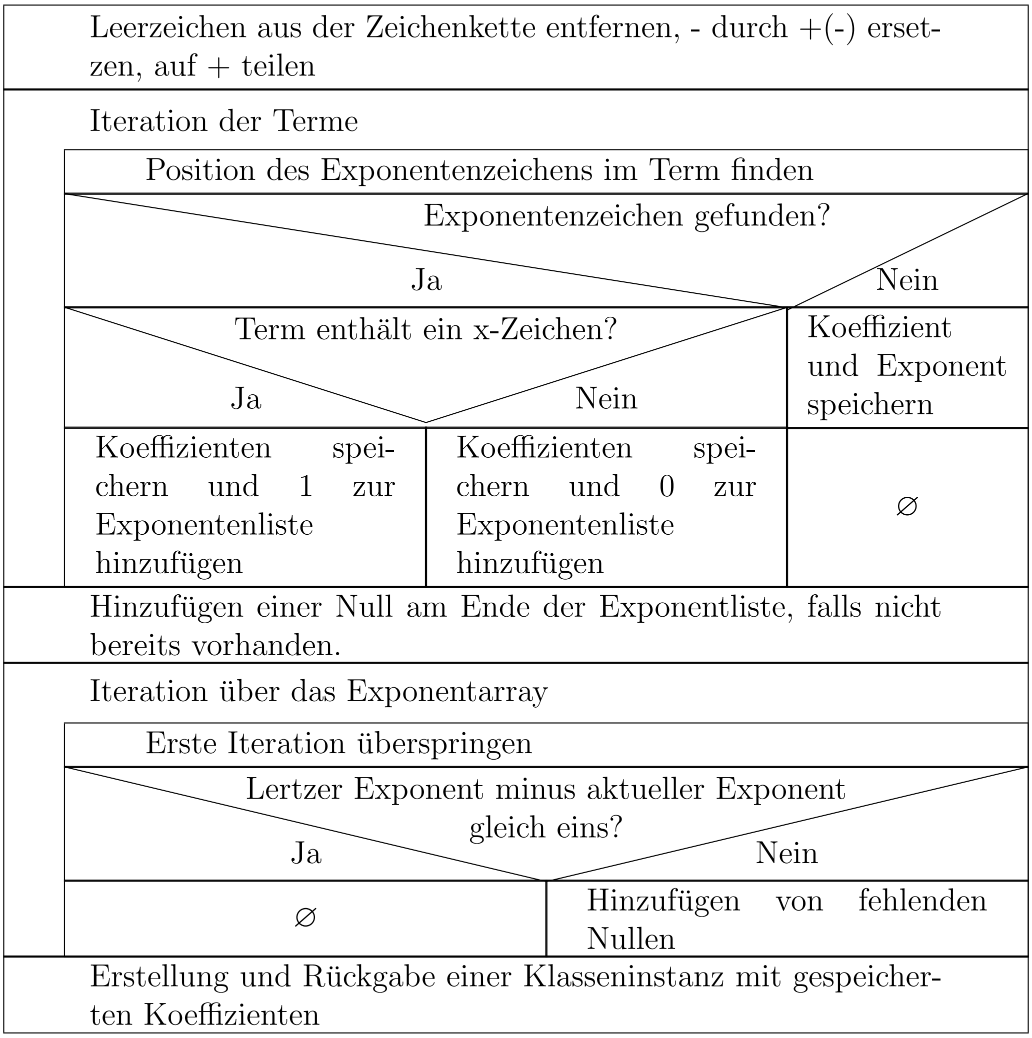


Abbildung 4: Struktogramm der Klassenmethode

In Abbildung 4 ist ein Struktogramm des Konvertierungsprozesses dargestellt. Die Funktion übernimmt die Eingabe und nimmt einige Formatierungen vor, zum Beispiel Leerzeichen entfernen und negative Vorzeichen durch ein Plus-Minus ersetzen. Dies ermöglichte es mir, die Zeichenfolge bei jedem Pluszeichen, in ein Array

[7] In einer dynamisch typisierten Sprache ist der Variablentyp implizit. Funktionen akzeptieren auch beliebige Typen, da sie nicht deklariert sind.

aufzuteilen. Dadurch konnte ich jeden einzelnen Term bearbeiten. Zum Beispiel wird $x^2 - 5x + 5$ in `['x^2', '-5x', '5']` aufgeteilt. Zwischen jedem Element im Array befindet sich ein implizites Pluszeichen. Meine Strategie bestand darin, das Array zu iterieren, jeden Koeffizienten und seinen Exponenten abzurufen und sie dann in zwei Arrays separat zu speichern. Dieser erste Teil funktioniert, gibt aber einige Polynomterme nicht, da die Koeffizienten gleich null sind, fehlen diese in dem Array. Aus diesem Grund musste ich auch die Exponenten speichern, so dass ich weiß, welche Koeffizienten fehlen. Mit Hilfe des Exponentenarrays kann ich herausfinden, an welcher Position im Array eine Null fehlt. Wenn das Exponentarray z.B. so aussehen sollte: `[6,5,3,2]`, kann man klar erkennen, das die Exponenten 4, 1 und 0 fehlen. Ich kann dann einfach die fehlenden Nullen an den entsprechenden Positionen hinzufügen.

## 4.2 Verarbeitung

Da die Verarbeitung in einem späteren Kapitel (Algorithmen) genauer analysiert wird, beschreibe ich an dieser Stelle exemplarisch, wie die Funktion `enumerate()` funktioniert, die sehr häufig während der Verarbeitung verwendet wird. Diese Art von Schleife wird existiert in den gängigsten Sprachen nicht.

In der ersten Schleife auf der rechten Seite wird, wie in anderen Sprachen, einfach über das Array iteriert. Andererseits iteriert die Funktion `enumerate()` nicht nur über das Array, sondern führt auch einen Index der Elemente ein. Es gibt ein Tupel mit diesen beiden Werten aus, wie in der zweiten Schleife rechts zu sehen ist. In Python ist es möglich, ein Tupel wie `a,b = (1,2)` zu teilen. Der Variablen `a` wird der Wert 1 und der Variablen `b` der Wert 2 zugewiesen, was als Entpacken bezeichnet wird. Es ist möglich, die gleiche Methode in der Schleife anzuwenden, wie unten gezeigt: Anstatt nur die Variable `x` zu schreiben, kann ich eine zweite Variable hinzufügen, die durch ein Komma getrennt ist. Die Variable `counter` beginnt immer bei Null, es sei denn, sie wird explizit durch einen Parameter anders definiert: `for counter, x in enumerate(array)`.

```
array = ['a','b','c']
# Schleife #1
for x in array:
    print(x)
    # 'a'
    # 'b'
    # 'c'

# Schleife #2
for x in enumerate(array):
    print(x)
    # (0, 'a')
    # (1, 'b')
    # (2, 'c')
```

## 4.3 Ausgabe

### 4.3.1 __str__ Funktion

Die `__str__()` methode wird verwendet, um dem Benutzer Informationen mit definierter Formatierung anzuzeigen. In diesem Fall sollte die `__str__`-Methode nicht die Koeffizienten anzeigen, sondern dem Benutzer die Funktion in der gewohnten

Formatierung anzeigen. Diese Methode wird automatisch ausgeführt, wenn `print()` für eine Instanz dieser Klasse verwendet wird.

### 4.3.2 Interaktion mit dem Program

Wie in vielen anderen Sprachen ist der Einstiegspunkt meiner Anwendung die Funktion `main()`. Obwohl dies in dieser Sprache nicht notwendig ist, habe ich mich entschieden, es trotzdem zu tun.Der folgende Codeblock ruft die Funktion `menu()` auf, wenn die Python-Datei direkt ausgeführt und nicht aus einer anderen Datei importiert wird. Dadurch kann ich die Ausführung der Menüfunktion verhindern, wenn die Klasse Polynomial von einer anderen Datei importiert wird.

```
if __name__ == "__main__":
    main()
```

Der Sinn dieser Funktion ist es, die Interaktion zwischen dem Benutzer und der Klasse ohne Programmierung zu ermöglichen. Die Funktion besteht aus einer Endlosschleife und einem vorherigen Aufbau. Ich habe ein Wörterbuch namens `menu_dict` verwendet, um dem Benutzer zu zeigen, welche Optionen verfügbar sind. Ich habe diesen Datentyp gewählt, weil er es mir ermöglicht, die Tastenkombinationen für jeden Vorgang einfach zu ändern. Wenn eine Verknüpfung geändert wird, benötigt sie keine zusätzlichen Änderungen im Code, da die „if"-Bedingungen in der Endlosschleife vergleichen, was der Benutzer mit dem entsprechenden Wert im Wörterbuch eingegeben hat, wie unten dargestellt:

```
if menu_dict[user_choice] == "Polynomial addition":
```

Jede durchgeführte Berechnung speichert nur die Darstellung, die das Speichern der vom Benutzer eingegebenen Funktionen erfordert, da es nicht benutzerfreundlich ist, mehr als eine Funktion zu plotten. Um die Funktionen zu speichern, habe ich auch ein Wörterbuch `user_functions` verwendet, das die Funktionen wie folgt speichert: Funktionszeichenkette: Zeiger im Speicher der Instanz. Die Funktionszeichenkette ist die gleiche Zeichenkette, die der Benutzer eingegeben hat, um eine Funktion hinzuzufügen, und ihr entsprechender Wert ist der Zeiger der Instanz auf die Position im Speicher. Dadurch entfällt die Notwendigkeit, eine Instanz jedes Mal neu zu erstellen, wenn der Benutzer die Grafik anzeigt.

# 5 Algorithmen

## 5.1 Grundlegende mathematische Operationen

### 5.1.1 Addition und Subtraktion

Die Programmierung einer Additions- oder Subtraktionsfunktion ist unkompliziert. Es ist wichtig zu beachten, dass diese beiden Funktionen die Berechnungen mit Klammern durchführen: $funktion_1 - (funktion_2)$. Vor der Durchführung der Berechnungen müssen ggf. einige Nullen hinzugefügt werden, da die Länge des Tupels, sprich wie viele Werte in dem Tupel stehen, das die Koeffizienten enthält, gleich sein muss. Die roten Zellen in Tabelle 2 sind die Nullen, die hinzugefügt werden müssen. Der Code prüft zuerst welches Polynom den höchsten Grad hat und fügt

Tabelle 2: Addition missing zeros

| Im code | Beispielsfunktion | Koeffizienten | | | |
|---|---|---|---|---|---|
| `self` | $5x^3 + 3$ | 5 | 0 | 0 | 3 |
| `other` | $8x + 10$ | 0 | 0 | 8 | 10 |

dann die erforderlichen Nullen hinzu. Der Gradunterschied zwischen den beiden Funktionen ist die Anzahl der fehlenden Nullen. Die Zeilen 149 und 151 sind im Wesentlichen gleich: Sie berechnet zuerst die Graddifferenz, dann wird das Tupel `(0,)` mit der Graddifferenz multipliziert. Es ist wichtig zu beachten, dass $4\cdot$ (0,) nicht 0, sondern `(0, 0, 0, 0)` ist. Dann addiert es die Koeffizienten an das Ende des Tupels, das mit Nullen gefüllt ist. Noch eine Bemerkung: `(a,) + (b,) = (a, b)`.

```
148 if self.degree > other.degree:
149     other.coeffs = (len(self) - len(other)) * (0,) + other.coeffs
150 elif self.degree < other.degree:
151     self.coeffs = (len(other) - len(self)) * (0,) + self.coeffs
```

Der letzte Block der Funktion durchläuft einfach die Koeffizienten der ersten Funktion (`self`). Die `enumerate()` Schleife behält einen Index des Objekts bei, das gerade durchlaufen wird. Dadurch kann ich auf den entsprechenden Koeffizienten im anderen Polynom zugreifen.

```
154 final = ()
155 for counter, coeff in enumerate(self.coeffs):
156     final += (coeff + other.coeffs[counter],)
157
158 return Polynomial(final)
```

Zeile 156 ist der einzige Unterschied zwischen der Additions- und der Subtraktionsfunktion. Unten ist dieselbe Zeile, jedoch in der Subtraktionsfunktion:

```
174 final += (coeff - other.coeffs[counter],)
```

Der einzige Unterschied ist das Plus- oder Minuszeichen. Natürlich sind ihre Namen unterschiedlich, beide sind „magische Methoden“, die implizit beim Addieren oder Subtrahieren von zwei Polynomen aufgerufen werden. Schließlich addieren oder subtrahieren sie beide die Koeffizienten des einen Polynoms mit denen des jeweiligen anderen und fügen das Ergebnis an das `final` Tupel an, das dann ausgegeben wird.

### 5.1.2 Horner-Schema

Das Horner-Schema ist ideal für die Durchführung der Polynomdivision, da es sehr übersichtlich und leicht zu programmieren ist. In der Abbildung 5 ist gezeigt, wie man das Horner-Schema auf Papier durchführt.

Das Horner-Schema befindet sich in der Funktion `__floordiv__` mit einem Parameter `other`, der eine Nullstelle einer Funktion ist. Diese „magische Funktion“ spezifiziert den //-Operator. Die Funktion iteriert über den Koeffizienten der Teilung unter Verwendung der Funktion `enumerate()`. Wie in Abbildung 5 dargestellt, wird der erste Koeffizient direkt zum Divisionsergebnis addiert. Da die Variable

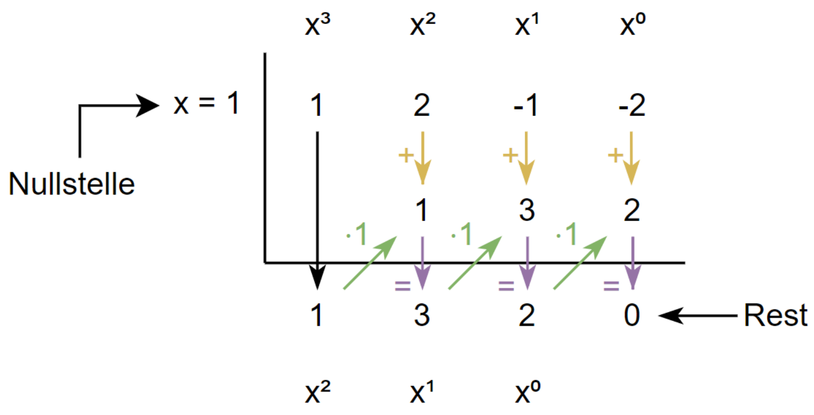


Abbildung 5: Horner Schema

`counter` bei Null beginnt und die Anzahl der Iterationen zählt, kann das Programm den Koeffizienten direkt an das Ergebnis-Tupel anfügen und zum nächsten Koeffizienten springen.

```
result = ()
for counter, coefficient in enumerate(self.coeffs):
    if counter == 0:
        result += (coefficient,)
    else:
        result += (coefficient + result[counter - 1] * other,)

return Polynomial(result[:-1])  # remove the rest
```

Wie in Abbildung 6 dargestellt, sind nach der ersten Iteration alle Schritte gleich: Die Nullstelle, sprich `other` mit dem vorherigen Koeffizienten im Ergebnistupel multiplizieren und das Ergebnis mit dem Koeffizienten addieren, der iteriert wird. Vor dem Erstellen und Zurückgeben der Instanz der Klasse ist es erforderlich, den Rest aus dem Tupel `result` zu entfernen, da es sich nicht um einen Koeffizienten handelt, sondern um den Rest der Division.

## 5.2 Nullstellen Finden

### 5.2.1 Newton-Verfahren

Der erste Schritt war das Schreiben einer Funktion, die die Nullstellen dieser Funktion berechnet. Um dies zu tun, habe ich die Newton-Methode verwendet, bei der $x_n$ eine zufällige ganze Zahl ist, vorzugsweise 0 oder 1, die exponentiell schnell zu einer Nullstelle konvergiert.

```
def solve_zero_newton(self):
    """
    :return: None if no zero found else zero
    """
    first_derivative = self.derivative()

    # this variable is the zero to be found
```

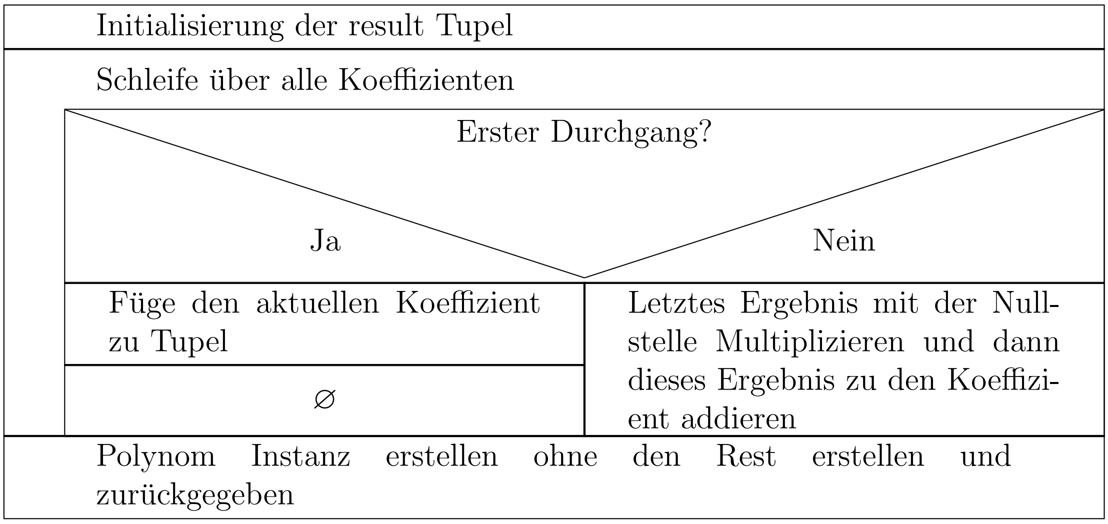


Abbildung 6: Struktogramm der Horner Schema Funktion

```
    y_cord = 0

    # if function has no zeros, the
    # the loop is never going to end
    iteration = 0
    # /!\ The rounding might be too big /!\
    while round(self(y_cord), 12) != 0:
        iteration += 1
        try:
            y_cord = y_cord - (self(y_cord) /
             ↪ first_derivative(y_cord))
        except ZeroDivisionError:
            # If unable to divide, move the starting x position
            y_cord += 1

        if iteration > 100:
            return None  # no solution

    return y_cord
```

Das erste, was diese Methode macht, ist, die Ableitung der Funktion zu erhalten, da sie für das Newton-Verfahren benötigt wird. Dann initialisiert es die Variable `y_cord`, die im Wesentlichen die Nullstelle enthält. Der Hauptteil dieser Funktion ist die **`while`** Schleife: Diese Schleife wird iterieren, bis $f(y\_cord) = 0$. Ich musste `y_cord` auf 12 Stellen nach dem Komma runden, um zu verhindern, dass die Schleife unnötig oft durchlaufen wird. Innerhalb der Schleife führe ich die Berechnung durch und weise das Ergebnis `y_cord` zu. Ich musste eine Fehlerprüfung gegen eine Division durch Null implementieren, da bei einer Funktion wie z.B. $x^2$ die Steigung der Tangente bei $x = 0$ 0 ist, was zu einer Division durch Null führen würde. In diesem Fall erhöht das Programm nur `y_cord` um 1 und wiederholt die Iteration.

Die Variable `iteration` wird verwendet, um die Schleife zu stoppen, wenn sie mehr als 100-mal wiederholt wird. In einigen Fällen, wie z.B. wenn eine Funktion keine Nullstellen hat, wird die Schleife nie enden, also muss ich sie manuell stoppen und `None` zurückgeben, was im Grunde bedeutet, dass keine Antwort gefunden wurde.

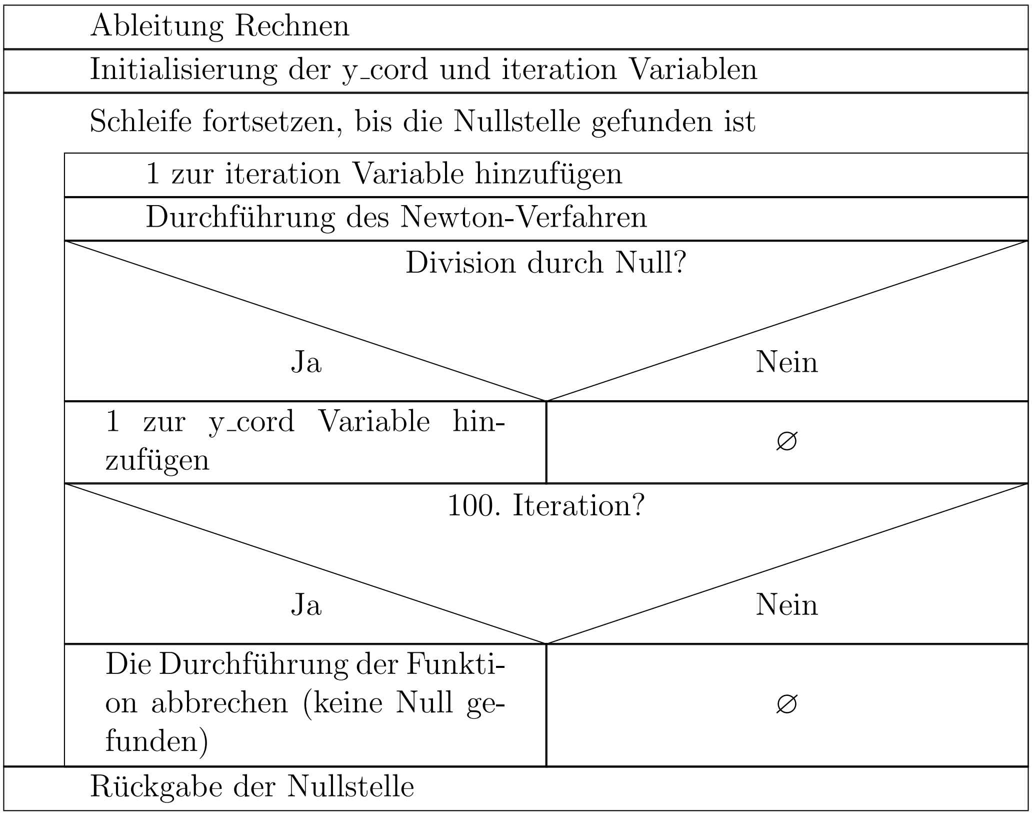


Abbildung 7: Struktogramm der Newton-Verfahrenfunktion

### 5.2.2 Kombination aus Horner-Schema und Newton-Verfahren

Da das Horner-Schema und das Newton-Verfahren nun in zwei Funktionen programmiert sind, ist es möglich, sie zu einem vollständigen Algorithmus zu kombinieren, der alle Nullstellen einer Funktion findet.

```
results = []
function = self
```

Die Nullstellen, die diese Methode findet, werden im `results`-Array gespeichert. Ich habe die Instanz auch in der lokalen `function` Variable gespeichert, anstatt sie direkt zu verwenden, weil die Variable `self` nicht geändert oder überschrieben werden darf. Die Variable `function` speichert das Resultat der Division dieser Funktion.

```
if self.degree < 1:
    return []
```

Um unnötige Verarbeitungen zu vermeiden, habe ich mich auch entschieden, zu prüfen, ob die Funktion ein Konstante ist, da sie in diesem Fall nie eine Nullstelle, bzw. bei $f(x) = 0$ eine unendliche Menge von ihnen hat. Trotzdem habe ich entschieden, dass es auch in diesem Fall ein leeres Array ausgibt.

```python
while str(function) != "1":
    # find a zero
    zero = function.solve_zero_newton()

    # solve_zero_newton will return None if
    # it didn't find a zero
    if zero is not None:
        results.append(round(zero, rounded))
        function = function // zero
    else:
        break

if single:
    return sorted(list(set(results)))

return sorted(results)
```

Die Schleife oben ist der Hauptteil der Funktion, wie in Abbildung 8 gezeigt. Die Schleife wird so lange iterieren, bis die Funktion gleich eins ist. Als erstes wird versucht, eine Nullstelle mit Newtons Methode zu finden. Ist dies nicht möglich, gibt die Funktion `None` zurück, d.h. ich muss überprüfen, ob die Variable `zero` gleich `None` ist und aus der Schleife ausbrechen, wenn sie es ist. Wenn die Variable nicht gleich `None` ist, dann wird das Ergebnis-Array um seinen Wert ergänzt und die Horner-Division durchgeführt. Dieser Vorgang wird solange wiederholt, bis die Funktion gleich 1 ist.

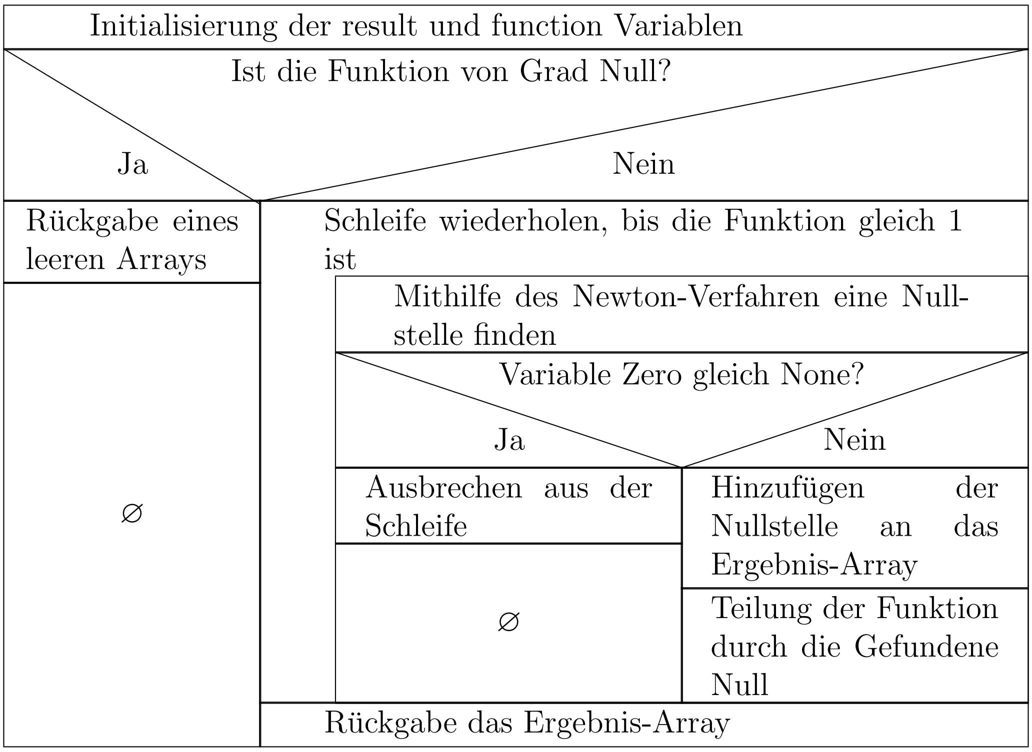


Abbildung 8: Struktogramm der Find_all_zeros Funktion

## 5.3 Ableitungsbezogene Funktionen

### 5.3.1 Ableitung

Die Ableitung ist sehr wichtig für die Analyse ganzrationaler Funktionen, da sie nicht nur die Steigung angibt, sondern mit ihr auch die Extrempunkte und Tangenten gefunden werden können. Daher ist eine Ableitungsmethode unerlässlich.

```
result = ()
for counter, coeff in enumerate(self.coeffs):
    exponent = self.degree - counter
    result += (exponent * coeff,)

return Polynomial(result[:-1])  # remove the last zero
```

Der erste Schritt besteht darin, ein Ergebnistupel zu initialisieren, das die Koeffizienten der Ableitung enthält. Die `enumerate()` Funktion gibt ein Tupel zurück, das die Variable und ihren Index im Tupel enthält. Dadurch kann ich den Exponenten dieses Koeffizienten abrufen, da der Grad der Funktion gleich bleibt, der Zähler jedoch weiter ansteigt. Die Tabelle 3 veranschaulicht dies.

Die jeweiligen Koeffizienten und Exponenten werden multipliziert und das Ergebnis dem Ergebnistupel hinzugerfügt. Ein Tupel in Python ist unveränderlich, deshalb muss man ein neues Tupel mit dem gleichen Namen erzeugen, um es verändern zu können, nach „Python Tuples“, W3Schools, 03. Dec 2018, `www.w3schools.com/python/python_tuples.asp`. Ich füge hier jedoch zwei Tupel hinzu, um ein drittes mit dem gleichen Namen wie zuvor zu erstellen. Der Grund, warum ich die Klammern und das Komma brauche, ist, dass ich das Ergebnis in ein Tupel konvertieren muss, bevor sie zusammengefügt werden, da es nicht möglich ist, einem Tupel etwas anderes als ein Tupel hinzuzufügen. Es ist im Wesentlichen ein Tupel mit einem einzigen Wert im Inneren.

Tabelle 3: Iterationsverfahren

| Zähler | Koeffizient $(4x^2 + 8x + 2)$ | Exponent | Ableitung |
|---|---|---|---|
| 0 | 4 | $2 - 0 = 2$ | $4 \cdot 2 = 8$ |
| 1 | 8 | $2 - 1 = 1$ | $8 \cdot 1 = 8$ |
| 2 | 2 | $2 - 2 = 0$ | $2 \cdot 0 = 0$ |

Bevor ich einen Wert zurückgebe, muss ich zuerst die letzte Null (rote Zelle) im Tupel entfernen, um die Werte nach rechts zu verschieben. Dann gebe ich damit eine Instanz der Polynomialklasse der Ableitung zurück.

### 5.3.2 Extrempunkte

Um die Extrempunkte zu finden, werden die erste und zweite Ableitung der Funktion benötigt. Die Schritte hierfür sind recht einfach, da ich nur meine bereits programmierten Funktionen verwenden muss, um sowohl die Ableitungen als auch die Nullstellen zu erhalten.

```
first_deriv = self.derivative()
second_deriv = first_deriv.derivative()
```

Ich beschloss, jede Art von Extrempunkt in einem anderen Array zu speichern. Auf diese Weise kann ich sie getrennt und leicht zugänglich speichern, wenn ich sie zurückschicke.

```
highpoints = []
lowpoints = []
turningpoints = []
```

Ich wollte mich zunächst mit den Hoch- und Tiefpunkten befassen, da sie mit Nullen der ersten Ableitung gefunden werden können. Ich muss nur die Funktion `find_all_zero()` aufrufen, um alle Nullstellen der ersten Ableitung zu erhalten, die ich dann mithilfe der zweiten Ableitung entsprechend an das Ergebnis-Array anfüge: die Nullstellen mit $f"(x) > 0$ sind die Tiefpunkte, die mit $f"(x) < 0$ sind Hochpunkte.

```
# High-, lowpoints
zeros = first_deriv.find_all_zero()
for zero in zeros:
    if second_deriv(zero) < 0:  # high point
        highpoints.append((zero, self(zero)))

    elif second_deriv(zero) > 0:  # low point
        lowpoints.append((zero, self(zero)))
```

Wie zuvor erwähnt, ist zum Auffinden der Wendepunkte die zweite Ableitung erforderlich. Alles, was man tun muss, ist, die Nullen der zweiten Ableitung zu finden und sie dann direkt an das Array anzuhängen. Bevor ich die Punkte an das Array anfügte, entschied ich mich, sie so zu formatieren, dass die Arrays wie folgt aussehen: [(x,y), (x,y)]. Die Abbildung 9 zeigt ein Strukturdiagramm der gesamten Funktion.

```
# turning points
zeros = second_deriv.find_all_zero()
for zero in zeros:
    turningpoints.append((zero, self(zero)))

return [lowpoints, turningpoints, highpoints]
```

### 5.3.3 Tangente

Die Ableitung einer Funktion ist ihre Steigung. Eine Tangente ist immer eine lineare Funktion und somit ist ihr Formel $y = m \cdot x + h$, wobei $x$ der Punkt ist, an dem die Tangente gesucht wird, $y$ der dazugehörige Funktionswert und die Steigung ist $m = f'(x)$.

```
first_derivative = self.derivative()

m = first_derivative(x_cord)

y = self(x_cord)

```

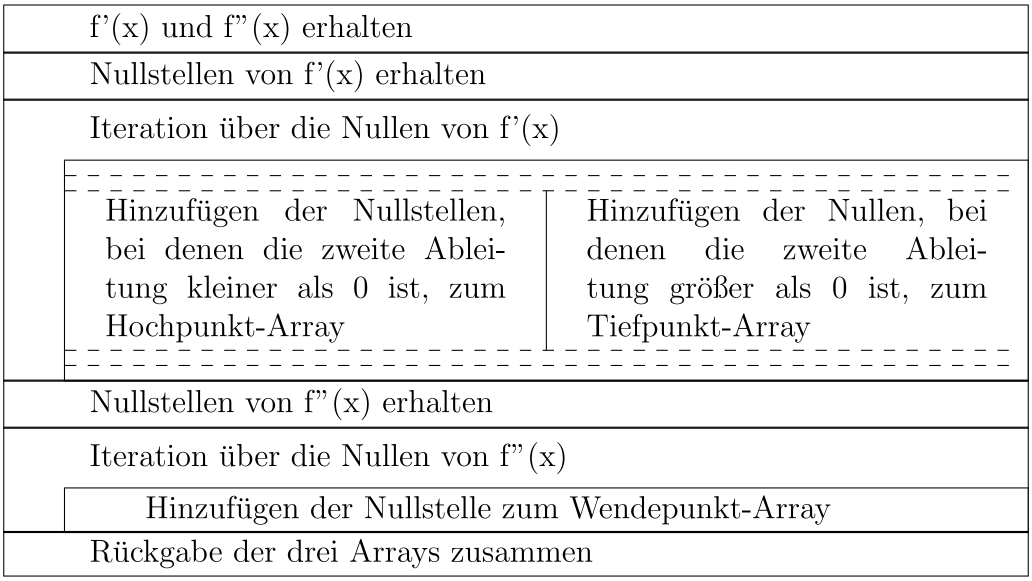


Abbildung 9: Extrempunkte funktion

```
261 h = -(m * x_cord - y)  # solve for h
262
263 return Polynomial((m, h))
```

Das erste, was zu tun ist, ist, die Ableitung der Funktion (Zeile 255) zu erhalten. Die $x$ Stelle, bei der die Tangente berechnet wird, wird als Parameter `x_cord` angegeben. Mit dieser Variable ist es möglich, $m$ zu berechnen, indem man $f'(x)$ (Zeile 257) sowie $y$ mit $f(x)$ berechnet. Durch Umstellen der Gleichung kann aus diesen Werten dann $h$ ermittelt werden. Anschließend gibt das Programm eine Instanz der Klasse mit $m$ und $h$ aus, da die Tangente eine Grad 1 Funktion ist, wobei $h$ die Konstante und $m$ der Grad 1 Koeffizient ist.

## 5.4 Integral

### 5.4.1 Integral

Das Integral ist genau das Gegenteil der Ableitung und somit sieht auch der Code ähnlich aus.

```
207 coeffs = self.coeffs + (0,)  # add a zero
```

Das erste, was ich getan habe, ist, eine Null am Ende der Koeffizienten hinzuzufügen. Dadurch kann ich die Koeffizienten nach links verschieben, wie in der Tabelle 4 dargestellt. Das Programm iteriert über die Koeffizienten mit der Schleife

Tabelle 4: Integral der Funktion $5x^2 + 10x + 30$

| | Koeffizienten | | | |
|---|---|---|---|---|
| Funktion | - | 5 | 10 | 30 |
| Stammfunktion | $\frac{5}{3}$ | $\frac{10}{2}$ | $\frac{30}{1}$ | 0 |

enumerate(). Es war erforderlich, eine Bedingung einzubauen, die überprüft, ob der Koeffizient gleich Null ist, da er sonst am Ende der Iteration zu einer Division durch Null führen würde, da der Exponent Null ist. Um die Division durchzuführen, wird der Exponent benötigt. Es ist möglich, ihn zu erhalten, indem man die Variable `counter`, die bei jeder Iteration größer wird, von der Länge des Koeffizienten-Tupels abzieht, die gleich bleibt. Es war nicht möglich, die `degree` Methode zu verwenden, da `coeffs` kein Polynomtyp, sondern ein Tupel ist und somit die Methode `degree` nicht hat.

```
result = ()
for counter, coeff in enumerate(coeffs):
    if coeff == 0:
        result += (0,)
        continue
    expo = len(coeffs) - counter - 1
    result += (coeff / expo,)

return Polynomial(result)
```

Der letzte Schritt vor dem Hinzufügen des berechneten Koeffizienten zum Ergebnistupel besteht darin, die Division mit dem Koeffizienten und dem Exponenten durchzuführen.

### 5.4.2 Flächeninhalt Rechnung

Die Berechnung der Fläche der Funktion war nicht so einfach, wie ich dachte, da die Flächen unter der x-Achse negativ ist, was zu einem falschen Ergebnis führen würde. Mein Ansatz, dieses Problem zu lösen, ist es, die Fläche zwischen jedem Nullpunkt innerhalb der vom Benutzer definierten Grenzen zu berechnen und ihre Beträge zu addieren. Ich habe eine Funktion mit zwei Parametern erstellt: untere Grenze und obere Grenze. Diese werden vom Benutzer definiert. Gebraucht sind natürlich die Nullstellen der Funktion, aber nur die innerhalb der vom Benutzer definierten Grenze, also iteriert das Programm über alle gefundenen Nullstellen und fügt sie nur dann einem `zeros` Tupel hinzu, wenn sie innerhalb der Grenzen liegen.

```
zeros = []
for zero in self.find_all_zero(single=True):
    if low_lim < zero < high_lim:
        zeros.append(zero)
```

Damit meine Idee funktioniert, musste ich die Grenzen zum Nullstellen-Array hinzufügen. Da das Nullstellen-Array von klein zu groß sortiert ist, habe ich die untere Grenze am Anfang und die obere Grenze am Ende hinzugefügt.

```
zeros.insert(0, low_lim)
zeros = zeros + [high_lim]
```

Wie in Abbildung 10 schematisiert, ist das Letzte, was zu tun ist, über das Nullstellen-Array zu iterieren. Da es nun auch die Grenzen enthält, ist es möglich, die Flächen zwischen Grenze und Nullstelle, bzw. zwischen den Nullstellen separat zu berechnen. Während der Iteration gibt es zwei Hauptvariablen, die sich ändern: die Nullstelle, die gerade iteriert wird, und die vorherige. Bei der ersten Iteration gibt es keine vorherige Nullstelle, d.h. das Programm muss nur die aktuelle Nullstelle als letzte Nullstelle definieren und zum nächsten Koeffizienten wechseln. Aus der zweiten Iteration berechnet das Programm den absoluten Wert des Bereichs zwischen dem aktuellen Nullpunkt/Grenzwert und dem letzten Wert und aktualisiert dann die `last` Variable.

```
result = 0
for counter, zero in enumerate(zeros):
    if counter == 0:
        last = zeros[counter]
        continue

    result += abs(self.integral()(zero) - (self.integral()(last)))
    last = zeros[counter]

return result
```

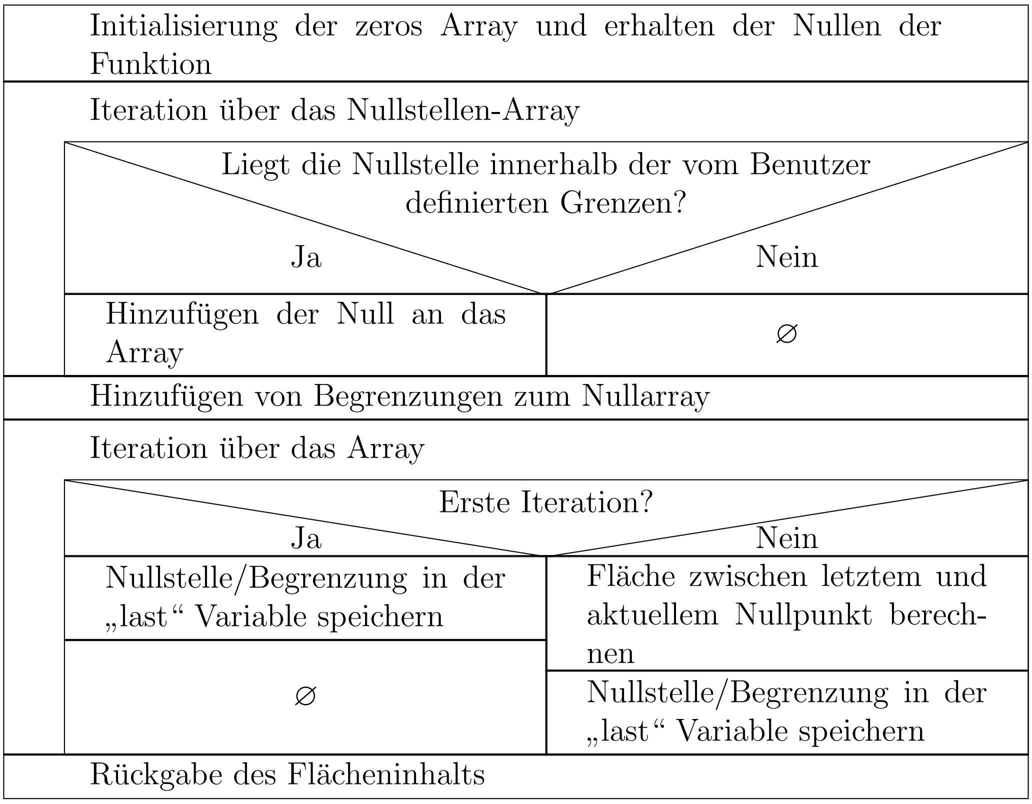


Abbildung 10: Struktogramm der Flächeninhalt Rechnung Funktion.

# 6 Das Testen

## 6.1 Modultest

Das Testen ist der wichtigste Teil des Entwicklungsprozesses eines Programms. Es erlaubt dem Entwickler, die Grenzen des Programms zu testen und Fehler zu finden. Es gibt ein Modul zum Testen namens Unittest, das sich in Pythons Standardbibliothek befindet. Dieses Modul ermöglicht es dem Entwickler, Einheiten seines Programms zu testen. Eine Einheit ist nur ein kleiner Teil des Programms, wie z.B. eine Funktion. Ich benutze dieses Modul jedoch, um mein ganzes Programm zu testen. Der Test muss insgesamt in einer anderen Datei erfolgen. Diese Datei muss sich im gleichen Verzeichnis wie das zu testende Skript befinden. Es gibt eine Namenskonvention, die besagt, immer „test“ vor das zu testende Objekt zu stellen.

```
import unittest

from main import Polynomial
```

Da sich der Testcode in einer anderen Datei befindet, ist es erforderlich, den Code zuerst aus der anderen Quelldatei zu importieren. Durch den Import des unittest Moduls erhält man die Testfunktionalität. Um das Testen einfach zu machen, habe ich mich entschieden, ein großes Array zu erstellen, das mehrere Arrays enthält, die eine Basisfunktion, deren Ableitung, Nullstellen, Integral und so weiter enthalten, wie in Tabelle 5 dargestellt. Der Vorteil bei dieser Art von Setup ist die einfache Erweiterbarkeit und leichte Lesbarkeit.

Tabelle 5: Dateiformat des Testarrays

| | | | | |
|---|---|---|---|---|
| $funktion_1$ | $ableitung_1$ | $integral_1$ | $zeros_1$ | $extrempunkte_1$ |
| $funktion_2$ | $ableitung_2$ | $integral_2$ | $zeros_2$ | $extrempunkte_2$ |
| ... | ... | ... | ... | ... |
| $funktion_n$ | $ableitung_n$ | $integral_n$ | $zeros_n$ | $extrempunkte_n$ |

Die Idee ist, die Berechnungsergebnisse meines Programms mit den vorher berechneten Ergebnissen einer anderen Methode zu vergleichen und zu sehen, ob sie übereinstimmen. Der Vorteil dieser Art von Test ist, dass bei einer Änderung in meinem Programm Fehler oder falsche Berechnungen leicht erkannt und korrigiert werden können.

# 7 Fazit

Ich persönlich denke, dass das Ziel erreicht wurde, weil es mir gelungen ist, zu programmieren, was ich ursprünglich wollte. Was mögliche Verbesserungen betrifft, so wird ein Computerprogramm nie fertig gestellt, es gibt immer etwas, das verbessert oder um neue Funktionen erweitert werden kann, aber mein Programm ist in seinem aktuellen Zustand funktionsfähig und tut genau das, was es eigentlich tun sollte. Es funktioniert wie vorgesehen, aber es fehlt die Fehlerprüfung auf der Interaktionsebene. Beim Schreiben dieses Programms ging ich davon aus, dass der Benutzer genau weiß, wie man mit dem Programm interagiert, und deshalb schließe ich die Fehlerprüfung nicht in den Interaktionsteil des Programms ein. Beispielsweise müssen die vom Benutzer eingegebenen Koeffizienten der Funktion absteigend sein, vom größten bis zum kleinsten Term, sonst stürzt das Programm ab.

Dieses Projekt hat mich gezwungen, die Programmierung aus einer ganz anderen Perspektive zu betrachten: ohne Bibliotheken. Bisher hatte ich nur mit vorhandenen Bibliotheken programmiert, so dass ich nicht über die Funktion einzelner Routinen nachdenken musste. So habe ich gelernt, das es viel schwieriger ist, Probleme zu lösen, die sich anfangs wirklich einfach anfühlten, wie z.B. das Finden von Nullstellen einer Funktion. Die Herausforderung hat mir Spaß gemacht, vor allem, weil ich gelernt habe, wie man einen Datentyp erstellt und mehr Erfahrung mit Klassenmethoden, Dekoratoren und objektorientierter Programmierung im Allgemeinen gesammelt habe. Die dabei gewonnenen Erfahrungen werden es mir ermöglichen, meine Reise in anderen objektorientierten Sprachen wie Java fortzusetzen und meiner Leidenschaft für die Informatik zu folgen. Außerdem hab ich beim Schreiben dieser Facharbeit gelernt, Dokumente mit LaTeXzu erstellen.

# Literaturverzeichnis


[1] https://stackoverflow.com/questions/2657268/whats-the-difference-between-compiled-and-interpreted-language [Stand: 23.02.2019]

[2] https://en.wikipedia.org/wiki/Programming_paradigm [Stand: 25.02.2019]

[3] https://en.wikibooks.org/wiki/Computer_Programming/Function_overloading [Stand: 16.12.2018]

[4] https://rubygarage.org/blog/clear-acceptance-criteria-and-why-its-important [Stand: 05.03.2019]

[5] https://stackoverflow.com/questions/7961363/removing-duplicates-in-lists [Stand: 14.12.2018]

[6] https://www.programiz.com/python-programming/methods/built-in/classmethod [Stand: 07.12.2018]

[7] https://www.w3schools.com/python/python_tuples.asp [Stand: 03.12.2018]

[8] https://www.geeksforgeeks.org/dunder-magic-methods-python/ [Stand: 10.03.2019]

[9] https://de.wikipedia.org/wiki/Objektorientierte_Programmierung [Stand: 24.02.2019]

[10] http://book.pythontips.com/en/latest/enumerate.html [Stand: 29.12.2018]

[11] https://www.python-course.eu/polynomial_class_in_python.php [Stand: 30.11.2018]

[12] https://stackoverflow.com/questions/19125722/adding-a-legend-to-pyplot-in-matplotlib-in-the-most-simple-manner-possible [Stand: 24.02.2019]

[13] https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages [Stand: 05.02.2019]

[14] https://de.wikipedia.org/wiki/Newton-Verfahren [Stand: 08.01.2019]

[15] https://en.wikipedia.org/wiki/Prototype [Stand: 10.04.2019]

# Anhang

# A Mainwindow.py

```python
# -*- coding: utf-8 -*-
# Python 3.6.6

# begin: 23 nov 2018

try:
    import matplotlib.pyplot as plt  # pip install matplotlib
except ModuleNotFoundError as exc:
    print("Matplotlib not found. Plotting will not work.")


class Polynomial:
    def __init__(self, coeffs: tuple):
        if type(coeffs) != tuple:
            raise TypeError("Only tuples are allowed.")

        self.coeffs = coeffs

    @classmethod
    def from_string(cls, string: str):
        """ 2nd init function (overloading in c++)"""
        coeffs = []  # contains coefficients
        exponent_list = []  # contains powers that are present

        string = string.replace(" ", "")
        string = string.replace("-", "+-")

        string = string.split("+")

        for term in string:
            # skip bits that are empty.
            # They are caused if the user enters
            # a function like -x
            if term == "":
                continue

            pow_sign_pos = term.find("^")

            # need to use eval to convert
            # fractions to float
            # if only x or constant
            if pow_sign_pos == -1:
                if 'x' in term:
                    term = term.replace("x", "")
                    term = term.replace("*", "")
                    term = "1" if term == "" else term  # support for x
                    term = "-1" if term == "-" else term  # support for -x
```

```python
                    coeffs.append(eval(term))
                    exponent_list.append(1)
                else:
                    # constants
                    coeffs.append(eval(term))
                    exponent_list.append(0)

            else:
                # bits from x^2
                base = term[:pow_sign_pos]
                base_multiplier = base.replace('x', '').replace('*', '')
                base_multiplier = 1 if base_multiplier == "" else base_multiplier
                base_multiplier = -1 if base_multiplier == "-" else base_multiplier
                exponent = term.replace(f"{base}^", "")
                exponent_list.append(int(exponent))
                coeffs.append(eval(str(base_multiplier)))

        # fix for the last zero (constant)
        if exponent_list[-1] != 0:
            exponent_list.append(0)
            coeffs.append(0)

        # add missing zeros in the coeffs list
        fixes = 0
        for position, iter in enumerate(exponent_list):
            if position == 0:
                continue

            previous = exponent_list[position - 1]

            if previous - iter != 1:
                for _ in range(previous - iter - 1):
                    coeffs.insert(position + fixes, 0)
                    fixes += 1

        return cls(tuple(coeffs))

    @property
    def degree(self):
        """ Use this to get the degree of the function """
        return len(self.coeffs) - 1

    def __repr__(self):
        """ must be = to how you created the instance """
        return f"{__class__.__name__}({self.coeffs})"

    def __str__(self):
        """ executed on print """
```

```python
        final_str = ""
        power = self.degree
        for coeff in self.coeffs:
            if float(coeff) == 0:
                power -= 1
                continue

            # important for formatting
            # don't replace the + 1 at the end without an x
            if power != 0:
                if float(coeff) == 1.0:
                    coeff = ""
                elif float(coeff) == -1.0:
                    coeff = "-"

            if power == 1:
                final_str += f"{coeff}x + "
            elif power == 0:
                final_str += f"{coeff} + "
            else:
                final_str += f"{coeff}x^{power} + "

            power -= 1

        final_str = final_str.replace("+ -", "- ")

        return "0" if final_str == "" else final_str[:-3]

    def __len__(self):
        """
        :return: the length of the coefficient tuple
        """
        return len(self.coeffs)

    def __call__(self, x: float):
        """
        Usage: instance(x) is equal to f(x)
        :param x: the x in f(x)
        :return: the image of x
        """
        result = 0
        for counter, coeff in enumerate(self.coeffs):
            result += coeff * x ** (self.degree - counter)

        return result

    def __add__(self, other):
        """
        :param other: Polynomial instance
        :return: Polynomial instance of addition
        """
```

```python
        # find the biggest polynomial to add missing zeros
        if self.degree > other.degree:
            other.coeffs = (len(self) - len(other)) * (0,) + other.coeffs
        elif self.degree < other.degree:
            self.coeffs = (len(other) - len(self)) * (0,) + self.coeffs

        # add the coefficients up
        final = ()
        for counter, coeff in enumerate(self.coeffs):
            final += (coeff + other.coeffs[counter],)

        return Polynomial(final)

    def __sub__(self, other):
        """
        :param other: Polynomial instance
        :return: Polynomial instance of addition
        """
        # find the biggest polynomial to add missing zeros
        if self.degree > other.degree:
            other = (len(self.coeffs) - len(other)) * (0,) + other.coeffs
        elif self.degree < other.degree:
            self.coeffs = (len(other) - len(self)) * (0,) + self.coeffs

        # add the coefficients up
        final = ()
        for counter, coeff in enumerate(self.coeffs):
            final += (coeff - other.coeffs[counter],)

        return Polynomial(final)

    def __floordiv__(self, other: float):
        """ Horner division
        :param other: (float/int) divisor
        :return: Polynomial instance of division without rest
        """
        result = ()
        for counter, coefficient in enumerate(self.coeffs):
            if counter == 0:
                result += (coefficient,)
            else:
                result += (coefficient + result[counter - 1] * other,)

        return Polynomial(result[:-1])  # remove the rest

    def derivative(self):
        """
```

```
        :return: derivative of given function
        """
        result = ()
        for counter, coeff in enumerate(self.coeffs):
            exponent = self.degree - counter
            result += (exponent * coeff,)

        return Polynomial(result[:-1])  # remove the last zero

    def integral(self):
        """
        :return: integral of given function
        """
        coeffs = self.coeffs + (0,)  # add a zero

        result = ()
        for counter, coeff in enumerate(coeffs):
            if coeff == 0:
                result += (0,)
                continue
            expo = len(coeffs) - counter - 1
            result += (coeff / expo,)

        return Polynomial(result)

    def calculate_area(self, low_lim: float, high_lim: float):
        """
        Known bugs: x^3-6x^2+9x has zero by 2.99 and 3.00 (actually double 3)
        That area between these zeros is negligible.
        :param low_lim: float / int
        :param high_lim: float / int
        :return: area of function
        """
        # add zeros that are inside bounds to zeros array
        zeros = []
        for zero in self.find_all_zero(single=True):
            if low_lim < zero < high_lim:
                zeros.append(zero)

        # add bounds to the array
        zeros.insert(0, low_lim)
        zeros = zeros + [high_lim]

        # Finally, calculate the area that is
        # between each zero and add it up
        result = 0
        for counter, zero in enumerate(zeros):
            if counter == 0:
                last = zeros[counter]
                continue
```

```python

            result += abs(self.integral()(zero) - (self.integral()(last)))
            last = zeros[counter]

        return result

    def calculate_tangent(self, x_cord: float):
        """
        :param x_cord: (int/float) the x cord where to calculate the tangent
        :return: Polynomial instance of the tangent
        """
        first_derivative = self.derivative()

        m = first_derivative(x_cord)

        y = self(x_cord)

        h = -(m * x_cord - y)  # solve for h

        return Polynomial((m, h))

    def get_extreme_points(self):
        """
        This method finds all extreme points ofa function
        :return: list of format [lowpoints, turningpoints, highpoints]
        """
        highpoints = []
        lowpoints = []
        turningpoints = []

        first_deriv = self.derivative()
        second_deriv = first_deriv.derivative()

        # High-, lowpoints
        zeros = first_deriv.find_all_zero()
        for zero in zeros:
            if second_deriv(zero) < 0:  # high point
                highpoints.append((zero, self(zero)))

            elif second_deriv(zero) > 0:  # low point
                lowpoints.append((zero, self(zero)))

        # turning points
        zeros = second_deriv.find_all_zero()
        for zero in zeros:
            turningpoints.append((zero, self(zero)))

        return [lowpoints, turningpoints, highpoints]
```

```python

    def solve_zero_newton(self):
        """
        :return: None if no zero found else zero
        """
        first_derivative = self.derivative()

        # this variable is the zero to be found
        y_cord = 0

        # if function has no zeros, the
        # the loop is never going to end
        iteration = 0
        # /!\ The rounding might be too big /!\
        while round(self(y_cord), 12) != 0:
            iteration += 1
            try:
                y_cord = y_cord - (self(y_cord) / first_derivative(y_cord))
            except ZeroDivisionError:
                # If unable to divide, move the starting x position
                y_cord += 1

            if iteration > 100:
                return None  # no solution

        return y_cord

    def find_all_zero(self, rounded=10, single=False):
        """
        :param single: whether to remove or leave the double zeros
        :param rounded: how many decimals to round the zeros to after comma
        :return: sorted list with all zeros
        """
        results = []
        function = self

        # if function is a constant
        # it has no zeros except
        # f(x)=0, which has infinite solutions
        if self.degree < 1:
            return []

        # The idea here is to find a zero of the function
        # using the Newton algorithm and then divide the
        # function by the zero until there is none left.
        # A function divided by its only zero will be equal to 1
        while str(function) != "1":
            # find a zero
            zero = function.solve_zero_newton()
```

```python

                # solve_zero_newton will return None if
                # it didn't find a zero
                if zero is not None:
                    results.append(round(zero, rounded))
                    function = function // zero
                else:
                    break

        if single:
            return sorted(list(set(results)))

        return sorted(results)

    @staticmethod
    def generate_x_array(lower_lim: int, upper_lim: int, values=1000):
        """
        This function calculates the x array
        For example: generate_x_array(0, 10, 6)
        will return:
        [0,2,4,6,8,10]
        :param lower_lim: int
        :param upper_lim: int
        :param values: values to be included in the array
        :return: an array with the x points
        """
        final_array = []
        difference = upper_lim - lower_lim
        increment = difference / (values - 1)

        for iter in range(values):
            final_array.append(lower_lim + increment * iter)

        return final_array


def main():
    sign = "\n>>> "

    # user function | format: (function: instance of class)
    user_functions = {}

    # Default plot limits
    lower_lim, upper_lim = -10, 10

    menu_dict = {"1": "Polynomial addition",
                 "4": "Calculate derivative",
                 "5": "Calculate tangent",
                 "6": "Calculate extreme points",
                 "7": "Calculate integral",
                 "8": "Calculate area",
```

```
                 "9": "Calculate zeros",
                 "+": "Add function",
                 "-": "Remove function",
                 ".": "Change plot limits",
                 "*": "Show functions",
                 "0": "Show Plot",
                 "h": "Show this help",
                 "e": "Exit"}

    # print the help menu once
    [print(f"[{x}]", menu_dict[x]) for x in menu_dict]

    """ MAIN PROGRAM LOOP """
    while True:
        user_choice = input(sign)
        if user_choice not in menu_dict:
            continue

        # """ ADDITION """
        if menu_dict[user_choice] == "Polynomial addition":
            first_poly = Polynomial.from_string(input(f"Enter first polynomial{sign}"))
            second_poly = Polynomial.from_string(input(f"Enter second polynomial{sign}"))
            print(f"Result (addition): {first_poly + second_poly}\n")

        # """ DERIVATIVE, TANGENT, EXTREME POINTS """
        elif menu_dict[user_choice] == "Calculate derivative":
            poly = Polynomial.from_string(input(f"Enter polynomial to derive{sign}"))
            print(f"Result (derivative): {poly.derivative()}\n")

        elif menu_dict[user_choice] == "Calculate tangent":
            tangent_function = Polynomial.from_string(input(f"Enter function to calculate tangent{sign}"))
            x_coord = float(input(f"Enter x point at which to calculate the tangent{sign}"))
            tan = tangent_function.calculate_tangent(x_coord)
            print(f"Result (tangent at x={x_coord}): {tan if str(tan) != '' else '0'}")

        elif menu_dict[user_choice] == "Calculate extreme points":
            point_function = Polynomial.from_string(input(f"Enter function to Extrempoints{sign}"))
            print(f"Result (Highpoints): {point_function.get_extreme_points()[2]}")
            print(f"Result (turningpuntke): {point_function.get_extreme_points()[1]}")
            print(f"Result (Lowpoints): {point_function.get_extreme_points()[0]}")

```

```python
        # """ INTEGRAL, AREA, ZEROS """
        elif menu_dict[user_choice] == "Calculate integral":
            poly = Polynomial.from_string(input(f"Enter polynomial to integrate{sign}"))
            print(f"Result (integration): {poly.integral()}\n")

        elif menu_dict[user_choice] == "Calculate area":
            # request functino from user and convert it to Polynomial
            area_function = Polynomial.from_string(input(f"Enter function{sign}"))

            # request limits of the area and convert them to integers
            low_bound, high_bound = input(f"Enter lower & upper limit separated by a comma{sign}").split(",")
            low_bound, high_bound = float(low_bound), float(high_bound)

            print(f"Result (area): {area_function.calculate_area(low_bound, high_bound)}")

        elif menu_dict[user_choice] == "Calculate zeros":
            zeros = Polynomial.from_string(input(f"Enter function{sign}"))
            print(zeros.find_all_zero())

        elif menu_dict[user_choice] == "Add function":
            function_to_add = input(f"Enter function to add:{sign}")
            user_functions[function_to_add] = Polynomial.from_string(function_to_add)

        elif menu_dict[user_choice] == "Remove function":  # TODO: FIX REMOVE FUNCTION THAT DOESNT EXIST
            function_to_remove = input(f"Enter function to remove:{sign}")
            user_functions.pop(function_to_remove)

        elif menu_dict[user_choice] == "Change plot limits":
            lower_lim, upper_lim = input("Enter upper and lower limit separated by a comma: ").split(",")
            lower_lim, upper_lim = float(lower_lim), float(upper_lim)

        elif menu_dict[user_choice] == "Show functions":
            print(user_functions)
            [print(f"[{counter}] {func}") for counter, func in enumerate(user_functions, 1)]

        elif menu_dict[user_choice] == "Show Plot":
            x_array = Polynomial.generate_x_array(lower_lim, upper_lim)

```

```python
            for func in user_functions:
                # generate y array for each function
                y_array = []
                for x in x_array:
                    # access the instance of that function in the dict
                    y_array.append(user_functions[func](x))

                plt.plot(x_array, y_array)

            plt.gca().legend([f"${str(function)}$" for function in user_functions])  # add function to legend
            plt.grid(True)
            plt.show()

        elif menu_dict[user_choice] == "Show this help":
            [print(f"[{x}]", menu_dict[x]) for x in menu_dict]

        elif menu_dict[user_choice] == "Exit":
            if input(f"Exit (y/n)? ") == "y":
                exit(0)


if __name__ == "__main__":
    main()
```

# B test_main.py

```python
import unittest

from main import Polynomial

testing = [
#   function,          degree, derivative,           integral,                      zeros,                       call(0), call(-10), extrempoints
    ["3x",                  1, "3",                  "1.5x^2",                      "[0]",                             0,  -30,      [[], [], []]                                                                  ],
    ["x",                   1, "1",                  "0.5x^2",                      "[0]",                             0,  -10,      [[], [], []]                                                                  ],
    ["-1/2x^2",             2, "-x",                 "-0.16666666666666666x^3",                          "[0, 0]",                              0,  -50,     [[], [], [(0, 0.0)]]                                                            ],
    ["x^2+0",               2, "2x",                 "0.3333333333333333x^3",                            "[0, 0]",                              0,  100,     [[(0, 0.0)], [], []]                                                             ],
    ["4x^5-2x^3+x",         5, "20x^4 - 6x^2 + 1",   "0.6666666666666666x^6 - 0.5x^4 + 0.5x^2",          "[0]",                                 0,  -398010, [[], [(-0.3872983346, -0.30596568433897803), (0, 0), (0.3872983346, 0.30596568433897803)], []]     ],
    ["-x^3-3/2x^2+8x-2", 3, "-3x^2 - 3.0x + 8", "-0.25x^4 - 0.5x^3 + 4.0x^2 - 2.0x",          "[-3.7655644371, 0.2655644371, 2.0]",          -2, 768,     [[(-2.2078251277, -16.212313244682942)], [(-0.5, -6.25)], [(1.2078251277, 3.712313244682943)]] ],
    ["x^3-6x^2+9x-2",     3, "3x^2 - 12x + 9",   "0.25x^4 - 2.0x^3 + 4.5x^2 - 2.0x",            "[0.2679491924, 2.0, 3.7320508076]",          -2, -1692,   [[(3.0, -2.0)], [(2.0, 0.0)], [(1.0, 2.0)]]                                  ],
```

```
["-4x^3+4x^2+8x",     3, "-12x^2 + 8x + 8",  "-x^4 + 1.3333333333333333x^3 + 4.0x^2",      "[-1.0, 0, 2.0]",      0,  4320,  [[(-0.5485837704, -2.5245212377635955)], [(0.3333333333, 2.9629629626518517)], [(1.215250437, 8.450447163689521)]]  ],
["x^3-6x^2+9x",       3, "3x^2 - 12x + 9",   "0.25x^4 - 2.0x^3 + 4.5x^2",     "[0, 2.9999996424, 3.0000003576]",    0,  -1690,   [[(3.0, 0.0)], [(2.0, 2.0)], [(1.0, 4.0)]]   ],
["-2x^2+x+1",         2, "-4x + 1",  "-0.6666666666666666x^3 + 0.5x^2 + x",     "[-0.5, 1.0]",    1,  -209,    [[], [], [(0.25, 1.125)]]   ],
["2x^5+4x^2-1",       5, "10x^4 + 8x",  "0.3333333333333333x^6 + 1.3333333333333333x^3 - x",  "[-1.1794043143, -0.5183777764, 0.4862225017]", -1, -199601,  [[(0, -1)], [(-0.5848035476, 0.23118268145783993)], [(-0.9283177667, 1.0682573024306086)]]   ],
["2x^4+3x^2-2x+1",    4, "8x^3 + 6x - 2",    "0.4x^5 + x^3 - x^2 + x",      "[]",     1,  20321,    [[(0.298035819, 0.6861842956155538)], [], []]   ],
["-4x^3+3x^2-2x+1",   3, "-12x^2 + 6x - 2",  "-x^4 + x^3 - x^2 + x",      "[0.6058295862]",     1,  4321,    [[], [(0.25, 0.625)], []]    ],
["4x^5-2x^3+x+1",     5, "20x^4 - 6x^2 + 1",  "0.6666666666666666x^6 - 0.5x^4 + 0.5x^2 + x",  "[-0.782968399]",    1,  -398009,  [[], [(-0.3872983346, 0.694034315661022), (0, 1), (0.3872983346, 1.305965684338978)], []]   ],
["3x^3-x^2+2x-1",     3, "9x^2 - 2x + 2",    "0.75x^4 - 0.3333333333333333x^3 + x^2 - x",    "[0.4598632694]",    -1, -3121, [[], [(0.1111111111, -0.7860082304736625)], []]   ],
["4x^4+2x^2-2",       4, "16x^3 + 4x",       "0.8x^5 + 0.6666666666666666x^3 - 2.0x",     "[-0.7071067812, 0.7071067812]",     -2, 40198, [[(0, -2)], [], []]   ],
```

```
    ["9x^9-8x^8+7x^7-6x^6+5x^5-4x^4+3x^3-2x^2+x+1", 9, "81x^8 - 64x^7 + 49x^6 - 36x^5 + 25x^4 - 16x^3 + 9x^2 - 4x + 1", "0.9x^10 - 0.8888888888888888x^9 + 0.875x^8 - 0.8571428571428571x^7 + 0.8333333333333334x^6 - 0.8x^5 + 0.75x^4 - 0.6666666666666666x^3 + 0.5x^2 + x", "[-0.3821609762]", 1, -9876543209, [[], [(0.4358419755, 1.2130755570561793)], []]]
]


class TestMain(unittest.TestCase):

    def test_repr(self):
        function = Polynomial.from_string("x^2")
        self.assertEqual(str(function.__repr__()), "Polynomial((1, 0, 0))")

    def test_len(self):
        function = Polynomial.from_string("x^2")
        self.assertEqual(function.__len__(), 3)

    def test_degree(self):
        for index, function in enumerate(testing):
            function = Polynomial.from_string(function[0])
            self.assertEqual(function.degree, testing[index][1])  # degree

    def test_derivative(self):
        for index, group in enumerate(testing):
            function = Polynomial.from_string(group[0])
            self.assertEqual(str(function.derivative()), testing[index][2])  # derivative

    def test_integral(self):
        for index, group in enumerate(testing):
            function = Polynomial.from_string(group[0])
            self.assertEqual(str(function.integral()), testing[index][3])  # Integral

    def test_zeros(self):
        for index, group in enumerate(testing):
            function = Polynomial.from_string(group[0])
            self.assertEqual(str(function.find_all_zero()), testing[index][4])  # zeros

    def test_call_zero(self):
        for index, group in enumerate(testing):
            function = Polynomial.from_string(group[0])
            self.assertEqual(function(0), testing[index][5])  # f(0)

    def test_call_neg_ten(self):
        for index, group in enumerate(testing):
```

```python
            function = Polynomial.from_string(group[0])
            self.assertEqual(function(-10), testing[index][6])  # f(-10)

    def test_extrempoints(self):
        for index, group in enumerate(testing):
            function = Polynomial.from_string(group[0])
            self.assertEqual(function.get_extreme_points(), testing[index][7])  # extrempoints

    def test_addition(self):
        function_1 = Polynomial.from_string("x^3+3x^2-5x-90")
        function_2 = Polynomial.from_string("-3x^3-3x^2-10x+45")
        self.assertEqual(str(function_1 + function_2), "-2x^3 - 15x - 45")

    def test_subtraction(self):
        function_1 = Polynomial.from_string("x^3+3x^2-5x-90")
        function_2 = Polynomial.from_string("-3x^3-3x^2-10x+45")
        self.assertEqual(str(function_1 - function_2), "4x^3 + 6x^2 + 5x - 135")

    def test_tangent(self):
        function_1 = Polynomial.from_string("2x^2")
        self.assertEqual(str(function_1.calculate_tangent(0)), "0")  # tangent at x=0

        function_2 = Polynomial.from_string("-3x^3-3x^2-10x+45")
        self.assertEqual(str(function_2.calculate_tangent(-10)), "-850x - 5655")  # tangent at x=-10

        function_3 = Polynomial.from_string("x^3+3x^2-5x-90")
        self.assertEqual(str(function_3.calculate_tangent(-5)), "40x + 85")  # tangent at x=-5

    def test_calculate_area(self):
        function_1 = Polynomial.from_string("x^3+5")
        self.assertEqual(function_1.calculate_area(-5, 9), 1829.3248196000754)  # -5,9
        self.assertEqual(function_1.calculate_area(-0, 0), 0)  # 0,0

        function_2 = Polynomial.from_string("-3x^3-3x^2-10x+45")
        self.assertEqual(function_2.calculate_area(-10, -5), 6756.25)  # -10,-5
        self.assertEqual(function_2.calculate_area(5, 10), 8056.25)  # 5,10

    def test_generate_x_array(self):
        self.assertEqual(Polynomial.generate_x_array(0, 10, 6), [0, 2, 4, 6, 8, 10])
        self.assertEqual(Polynomial.generate_x_array(0, 20, 10),
```

```
                                [0, 2.2222222222222223, 4.444444444444445,
                                ↪  6.666666666666667, 8.88888888888889,
                                11.11111111111111, 13.333333333333334,
                                 ↪  15.555555555555557, 17.77777777777778,
                                 ↪  20])


if __name__ == "__main__":
    unittest.main(verbosity=2)
```