Ein Einsteigerhandbuch zur objektorientierten Programmierung (OOP) mit Python

Programmieren ist eine Kunst. Und wie in der Kunst ist die Auswahl der richtigen Pinsel und Farben wichtig, um die besten Werke zu schaffen. Die objektorientierte Programmierung mit Python ist eine solche Fähigkeit.

Die Wahl der richtigen Programmiersprache ist ein entscheidender Teil eines jeden Projekts und es kann entweder zu einer flüssigen und angenehmen Entwicklung oder zu einem kompletten Albtraum führen. Daher wäre es am besten, wenn du die Sprache verwendest, die am besten zu deinem Anwendungsfall passt.

Das ist der Hauptgrund, objektorientiertes Programmieren in Python zu lernen, welches auch eine der beliebtesten Programmiersprachen ist.

Lasst uns lernen!

Ein Beispiel Python Programm

Bevor wir in die Materie eintauchen, wollen wir eine Frage stellen: Hast du jemals ein Python-Programm wie das untenstehende geschrieben?

```
secret_number = 20

while True:
   number = input('Guess the number: ')

   try:
       number = int(number)
   except:
       print('Sorry that is not a number')
       continue

   if number != secret_number:
       if number > secret_number:
```

```
            print(number, 'is greater than the secret number')

        elif number < secret_number:
            print(number, 'is less than the secret number')
    else:
        print('You guessed the number:', secret_number)
        break
```

Dieser Code ist ein einfacher Zahlenschätzer. Versuche ihn in eine Python Datei zu kopieren und führe ihn in deinem System aus. Es erfüllt perfekt seinen Zweck.

Aber hier kommt ein großes Problem: Was wäre, wenn wir dich bitten würden, ein neues Feature zu implementieren? Es könnte etwas Einfaches sein – zum Beispiel:

„Wenn die Eingabe ein Vielfaches der Geheimzahl ist, gib dem Benutzer einen Hinweis."

Das Programm würde schnell komplex und schwer werden, wenn du die Anzahl der Features und damit die Gesamtzahl der verschachtelten Conditionals erhöhst.

Das ist genau das Problem, das die objektorientierte Programmierung zu lösen versucht.

Voraussetzungen, um Python OOP zu lernen

Bevor du in die objektorientierte Programmierung einsteigst, empfehlen wir dir, die Grundlagen von Python zu beherrschen.

Die Klassifizierung von Themen, die als „grundlegend" gelten, kann schwierig sein. Aus diesem Grund haben wir einen Spickzettel mit den wichtigsten Konzepten erstellt, die du zum Erlernen der objektorientierten Programmierung in Python benötigst.

- **Variable**: Symbolischer Name, der auf ein bestimmtes **Objekt** zeigt (wir werden im Laufe des Artikels sehen, was Objekte bedeuten).
- **Arithmetische Operatoren**: Addition (+), Subtraktion (-), Multiplikation (*), Division (/), ganzzahlige Division (//), modulo (%).

- **Eingebaute Datentypen**: Numerisch (Integer, Float, Komplex), Sequenzen (Strings, Listen, Tupel), Boolesch (True, False), Dictionaries und Sets.
- **Boolesche Ausdrücke**: Ausdrücke, bei denen das Ergebnis **True** oder **False**
- **Conditional**: Wertet einen booleschen Ausdruck aus und führt abhängig vom Ergebnis einen Prozess aus. Wird durch **if/else** Anweisungen gehandhabt.
- **Schleife**: Wiederholte Ausführung von Codeblöcken. Es kann sich um **for** oder **while** Schleifen handeln.
- **Funktionen**: Block von organisiertem und wiederverwendbarem Code. Du erstellst sie mit dem Keyword **def**.
- **Argumente**: Objekte, die an eine Funktion übergeben werden. Zum Beispiel: `sum([1, 2, 4])`
- **Führe ein Python-Skript aus**: Öffne ein Terminal oder eine Kommandozeile und gib „python <Name der Datei>“ ein.
- **Öffne eine Python-Shell**: Öffne ein Terminal und tippe „`python`“ oder „`python3`“ ein, abhängig von deinem System.

Jetzt, wo du diese Konzepte kristallklar hast, kannst du mit dem Verständnis der objektorientierten Programmierung fortfahren.

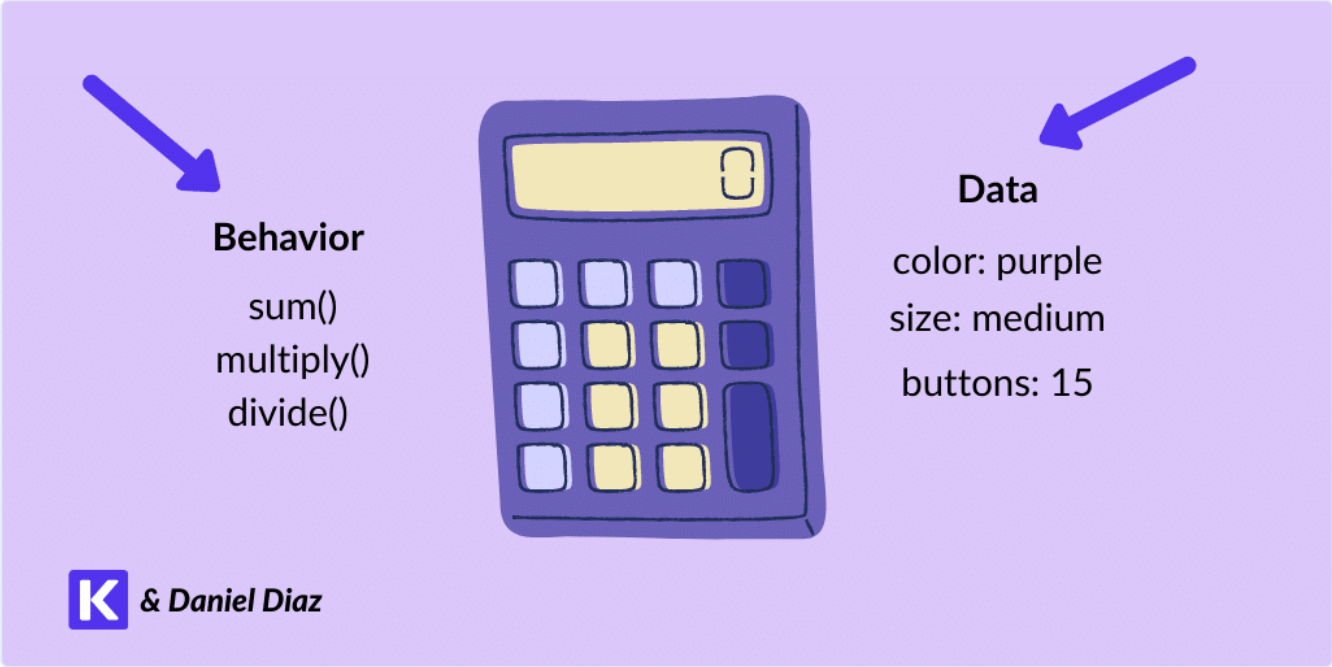
Was ist objektorientiertes Programmieren in Python?

Objektorientierte Programmierung (OOP) ist ein Programmierparadigma, in dem wir über komplexe Probleme als Objekte denken können.

Ein Paradigma ist eine Theorie, die die Basis für das Lösen von Problemen liefert

Wenn wir also über OOP sprechen, beziehen wir uns auf eine Reihe von Konzepten und Mustern, die wir verwenden, um Probleme mit Objekten zu lösen.

Ein Objekt in Python ist eine einzelne Sammlung von Daten (Attribute) und Verhalten (Methoden). Du kannst dir Objekte als reale Dinge vorstellen, die dich umgeben. Betrachte zum Beispiel Taschenrechner:



— Ein Taschenrechner kann ein Objekt sein.

Wie du vielleicht bemerkst, sind die Daten (Attribute) immer Substantive, während die Verhaltensweisen (Methoden) immer Verben sind.

Diese Aufteilung ist das zentrale Konzept der objektorientierten Programmierung. Du baust Objekte, die Daten speichern und bestimmte Arten von Funktionalität enthalten.

Warum verwenden wir objektorientierte Programmierung in Python?

OOP erlaubt es dir, sichere und zuverlässige Software zu erstellen. Viele Python Frameworks und Bibliotheken nutzen dieses Paradigma, um ihre Codebasis aufzubauen. Einige Beispiele sind Django, Kivy, Pandas, NumPy und TensorFlow.

Schauen wir uns die Hauptvorteile der Verwendung von OOP in Python an.

Vorteile von Python OOP

Die folgenden Gründe werden dich dazu bringen, dich für die Verwendung von objektorientierter Programmierung in Python zu entscheiden.

Alle modernen Programmiersprachen verwenden OOP

Dieses Paradigma ist sprachunabhängig. Wenn du OOP in Python lernst, kannst du es in den folgenden Sprachen verwenden:

- Java
- PHP (lies unbedingt den Vergleich zwischen PHP und Python)
- Ruby
- Javascript
- C#
- Kotlin

Alle diese Sprachen sind entweder nativ objektorientiert oder beinhalten Optionen für objektorientierte Funktionalität. Wenn du eine dieser Sprachen nach Python lernen willst, wird es einfacher sein – du wirst viele Ähnlichkeiten zwischen den Sprachen finden, die mit Objekten arbeiten.

OOP erlaubt es dir, schneller zu programmieren

Schneller zu kodieren bedeutet nicht, weniger Zeilen Code zu schreiben. Es bedeutet, dass du mehr Funktionen in kürzerer Zeit implementieren kannst, ohne die Stabilität eines Projekts zu gefährden.

Objektorientiertes Programmieren erlaubt es dir, Code wiederzuverwenden, indem du Abstraktion implementierst. Dieses Prinzip macht deinen Code prägnanter und lesbarer.

Wie du vielleicht weißt, verbringen Programmierer viel mehr Zeit damit, Code zu lesen als ihn zu schreiben. Es ist der Grund, warum Lesbarkeit immer wichtiger ist, als Features so schnell wie möglich herauszubekommen.

— Produktivität sinkt mit nicht lesbarem Code

Du wirst später mehr über das Abstraktionsprinzip erfahren.

OOP hilft dir, Spaghetti Code zu vermeiden

Erinnerst du dich an das Zahlenschätzer-Programm am Anfang dieses Artikels?

Wenn du weiterhin Funktionen hinzufügst, wirst du in Zukunft viele verschachtelte **if**-Anweisungen haben. Dieses Gewirr von endlosen Codezeilen nennt man Spaghetti-Code, und du solltest es so weit wie möglich vermeiden.

OOP gibt uns die Möglichkeit, die gesamte Logik in Objekten zu komprimieren und somit lange Stücke von verschachtelten **if's** zu vermeiden.

OOP verbessert deine Analyse von jeder Situation

Sobald du etwas Erfahrung mit OOP gesammelt hast, wirst du in der Lage sein, Probleme als kleine und spezifische Objekte zu betrachten.

Dieses Verständnis führt zu einer schnellen Projektinitialisierung.

Strukturierte Programmierung vs. Objekt-orientierte Programmierung

Die strukturierte Programmierung ist das von Anfängern am häufigsten verwendete Paradigma, weil es der einfachste Weg ist, ein kleines Programm zu bauen.

Es geht darum, ein Python-Programm sequentiell ablaufen zu lassen. Das heißt, du gibst dem Computer eine Liste von Aufgaben und führst sie dann von oben nach unten aus.

Schauen wir uns ein Beispiel für strukturiertes Programmieren mit einem Coffee Shop Programm an.

```
small = 2
regular = 5
big = 6

user_budget = input('What is your budget? ')

try:
   user_budget = int(user_budget)
except:
   print('Please enter a number')
   exit()

if user_budget > 0:
   if user_budget >= big:
       print('You can afford the big coffee')
       if user_budget == big:
           print('It\'s complete')
       else:
           print('Your change is', user_budget - big)
   elif user_budget == regular:
       print('You can afford the regular coffee')
```

```
        print('It\'s complete')
    elif user_budget >= small:
        print('You can buy the small coffee')
        if user_budget == small:
            print('It\'s complete')
        else:
            print('Your change is', user_budget - small)
```

Der obige Code agiert wie ein Kaffeehausverkäufer. Er fragt dich nach einem Budget und „verkauft“ dir dann den größten Kaffee, den du kaufen kannst.

Versuche, es im Terminal auszuführen. Er wird Schritt für Schritt ausgeführt, abhängig von deinen Eingaben.

Dieser Code funktioniert perfekt, aber wir haben drei Probleme:

1. Es hat eine Menge an wiederholter Logik.
2. Es verwendet viele verschachtelte **if**-Bedingungen.
3. Es wird schwer zu lesen und zu ändern sein.

OOP wurde als Lösung für all diese Probleme erfunden.

Schauen wir uns das obige Programm an, das mit OOP implementiert wurde. Mach dir keine Sorgen, wenn du es noch nicht verstehst. Es dient nur dazu, strukturierte Programmierung und objektorientierte Programmierung zu vergleichen.

```
class Coffee:
        # Constructor
        def __init__(self, name, price):
                self.name = name
                self.price = float(price)
        def check_budget(self, budget):
```

```
                # Check if the budget is valid
                if not isinstance(budget, (int, float)):
                        print('Enter float or int')
                        exit()
                if budget < 0:
                    print('Sorry you don\'t have money')
                    exit()
        def get_change(self, budget):
                return budget - self.price

        def sell(self, budget):
                self.check_budget(budget)
                if budget >= self.price:
                        print(f'You can buy the {self.name} coffe
                        if budget == self.price:
                                print('It\'s complete')
                        else:
                                print(f'Here is your change {self

                        exit('Thanks for your transaction')
```

Hinweis: Alle folgenden Konzepte werden im weiteren Verlauf des Artikels erklärt.

Der obige Code repräsentiert eine **Klasse** namens „Coffee“. Es hat zwei Attribute – „name“ und „price“ – und beide werden in den Methoden verwendet. Die primäre Methode ist „sell“, die die gesamte Logik verarbeitet, die benötigt wird, um den Verkaufsprozess abzuschließen.

Wenn du versuchst, diese Klasse auszuführen, wirst du keine Ausgabe erhalten. Das liegt vor allem daran, dass wir nur das „Template“ für die Kaffees deklarieren, nicht die Kaffees selbst.

Lass uns diese Klasse mit dem folgenden Code implementieren:

```
small = Coffee('Small', 2)
regular = Coffee('Regular', 5)
big = Coffee('Big', 6)

try:
   user_budget = float(input('What is your budget? '))
except ValueError:
   exit('Please enter a number')

for coffee in [big, regular, small]:
   coffee.sell(user_budget)
```

Hier erzeugen wir **Instanzen** oder Kaffee-Objekte der Klasse „Coffee“ und rufen dann die „Sell“-Methode jedes Kaffees auf, bis der Benutzer sich eine Option leisten kann.

Mit beiden Ansätzen erhalten wir die gleiche Ausgabe, aber wir können die Funktionalität des Programms mit OOP viel besser erweitern.

Unten ist eine Tabelle, die objektorientierte Programmierung und strukturierte Programmierung vergleicht:

OOP	**Strukturierte Programmierung**
Leichter zu warten	Schwer zu warten
Don’t Repeat Yourself (DRY) Ansatz	Wiederholter Code an vielen Stellen
Kleine Codestücke, die an vielen Stellen wiederverwendet werden	Eine große Menge an Code an wenigen Stellen
Objektansatz	Block Code Ansatz
Einfachere Fehlersuche	Schwerer zu beheben
Großer Lernaufwand	Einfachere Lernkurve
Einsatz in großen Projekten	Optimiert für einfache Programme

Um den Paradigmenvergleich abzuschließen:

- Keines der beiden Paradigmen ist perfekt (OOP kann überwältigend sein, wenn man es in einfachen Projekten einsetzt).
- Dies sind nur zwei Möglichkeiten, ein Problem zu lösen; es gibt noch andere.
- OOP wird in großen Codebases verwendet, während strukturierte Programmierung hauptsächlich für einfache Projekte gedacht ist.

Lass uns mit den eingebauten Objekten in Python weitermachen.

Alles ist ein Objekt in Python

Wir verraten dir ein Geheimnis: Du hast die ganze Zeit OOP benutzt, ohne es zu bemerken.

Selbst wenn du andere Paradigmen in Python verwendest, benutzt du immer noch Objekte, um fast alles zu tun.

Das liegt daran, dass in Python alles ein Objekt ist.

Erinnere dich an die Definition von Objekten: Ein Objekt in Python ist eine einzelne Sammlung von Daten (Attributen) und Verhalten (Methoden).

Das passt zu jedem Datentyp in Python.

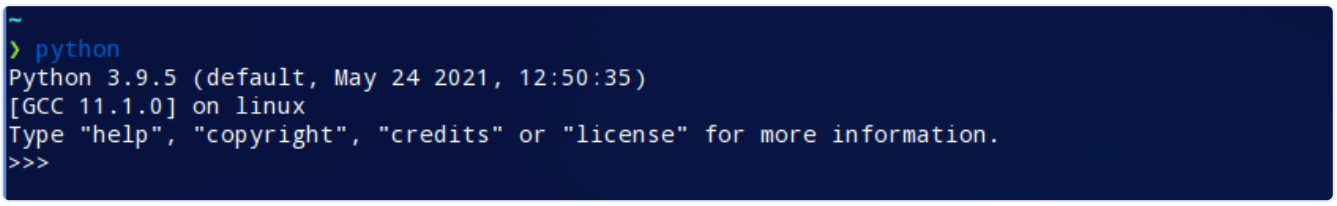
Ein String ist eine Sammlung von Daten (Zeichen) und Verhaltensweisen (**upper(), lower()**, etc..). Das Gleiche gilt für **Integer, Floats, Booleans, Listen** und Dictionaries.

Bevor wir fortfahren, lass uns die Bedeutung von Attributen und Methoden besprechen.

Attribute und Methoden

Attribute sind interne **Variablen** innerhalb von Objekten, während Methoden **Funktionen** sind, die ein bestimmtes Verhalten erzeugen.

Lass uns eine einfache Übung in der Python-Shell machen. Du kannst es öffnen, indem du `python` oder `python3` in dein Terminal eingibst.



— Python-Shell

Nun wollen wir mit der Python-Shell arbeiten, um Methoden und Typen zu entdecken.

```
>>> kinsta = 'Kinsta, Premium Application, Database, and Managed W
>>> kinsta.upper()
'KINSTA, PREMIUM APPLICATION, DATABASE, AND MANAGED WORDPRESS HOST
```

In der zweiten Zeile rufen wir eine String-Methode auf, **upper()**. Sie gibt den Inhalt des Strings in Großbuchstaben zurück. Die ursprüngliche Variable wird dadurch aber nicht verändert.

```
>>> kinsta
'Kinsta, Premium Application, Database, and Managed WordPress host
```

Lass uns auf wertvolle Funktionen bei der Arbeit mit Objekten eingehen.

Die **type()**-Funktion erlaubt es dir, den Typ eines Objekts zu erhalten. Der „Typ“ ist die Klasse, zu der das Objekt gehört.

```
>>> type(kinsta)
# class 'str'
```

Die **dir()** Funktion gibt alle Attribute und Methoden eines Objekts zurück. Lass uns es mit der **kinsta**-Variable ausprobieren.

```
>>> dir(kinsta)
['__add__', '__class__',  ..... 'upper', 'zfill']
```

Versuche nun, einige der versteckten Attribute dieses Objekts zu drucken.

```
 >>> kinsta.__class__ # class ‘str’ e>
```

Dies wird die Klasse ausgeben, zu der das Objekt **kinsta** gehört. Wir können also sagen, dass das einzige, was die **Typ-**funktion zurückgibt, das __**class**__ Attribut eines Objekts ist.

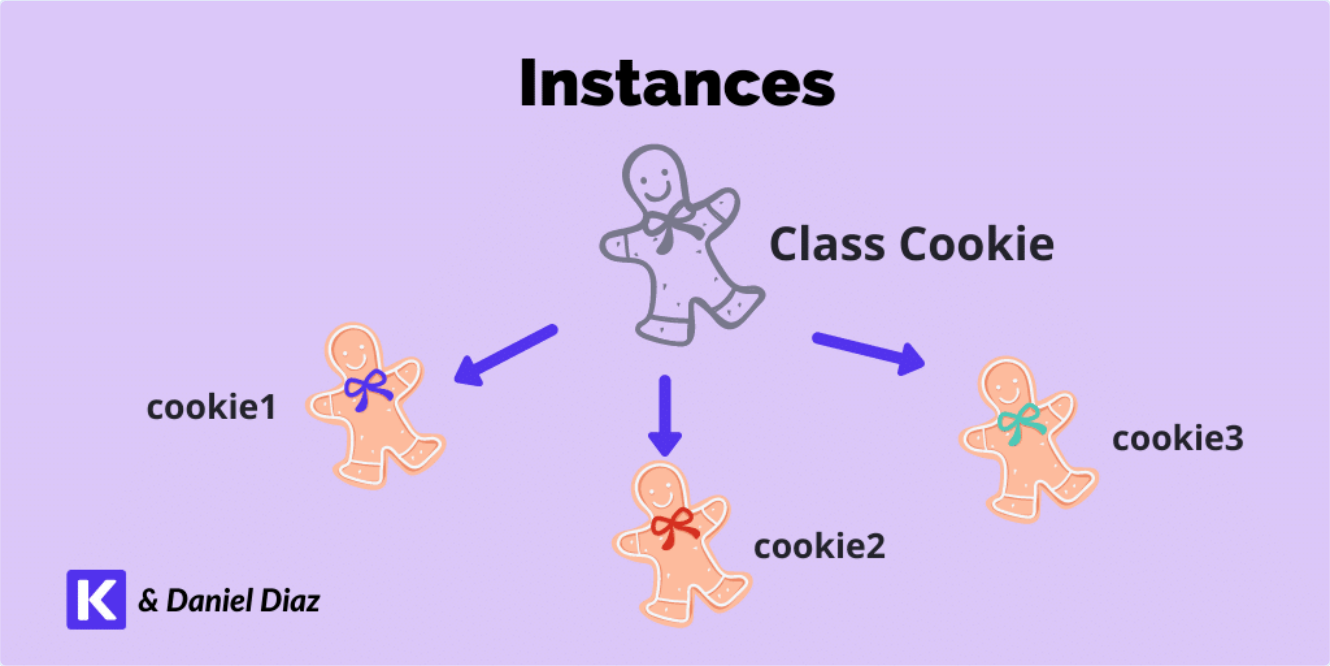
Du kannst mit allen Datentypen experimentieren und alle ihre Attribute und Methoden direkt auf dem Terminal entdecken. Du kannst mehr über die eingebauten Datentypen in der offiziellen Dokumentation erfahren.

Dein erstes Objekt in Python

Eine **Klasse** ist wie ein Template. Sie erlaubt es dir, eigene Objekte zu erstellen, die auf den von dir definierten Attributen und Methoden basieren.

Du kannst es dir wie eine **Ausstechform** vorstellen, die du modifizierst, um die perfekten Kekse (Objekte, nicht Tracking-Cookies) zu backen, mit definierten Eigenschaften: Form, Größe, und mehr.

Auf der anderen Seite haben wir **Instanzen**. Eine Instanz ist ein individuelles Objekt einer Klasse, das eine eindeutige Speicheradresse hat.



— Instanzen in Python

Jetzt, wo du weißt, was Klassen und Instanzen sind, lass uns welche definieren!

Um eine **Klasse** in Python zu definieren, benutzt du das Keyword class, gefolgt von ihrem Namen. In diesem Fall wirst du eine Klasse namens **Cookie** erstellen.

Hinweis: In Python verwenden wir die Konvention der Groß- und Kleinschreibung, um Klassen zu benennen.

```
class Cookie:
 pass
```

Öffne deine Python-Shell und gib den obigen Code ein. Um eine Instanz einer Klasse zu erzeugen, gibst du einfach den Namen der Klasse und die Klammern dahinter ein. Es ist der gleiche Prozess wie der Aufruf einer Funktion.

```
cookie1 = Cookie()
```

Glückwunsch – du hast gerade dein erstes Objekt in Python erstellt! Du kannst seine id und seinen Typ mit dem folgenden Code überprüfen:

```
id(cookie1)
140130610977040 # Unique identifier of the object

type(cookie1)
<class '__main__.Cookie'>
```

Wie du sehen kannst, hat dieses Cookie einen eindeutigen Bezeichner im Speicher und sein Typ ist **Cookie**.

Du kannst auch mit der Funktion **isinstance()** überprüfen, ob ein Objekt eine Instanz einer Klasse ist.

```
isinstance(cookie1, Cookie)
# True
isinstance(cookie1, int)
# False
isinstance('a string', Cookie)
# False
```

Konstruktor-Methode

Die **__init__()** Methode wird auch „Konstruktor“ genannt. Sie wird von Python jedes Mal aufgerufen, wenn wir ein Objekt instanziieren.

Der Konstruktor erzeugt den Anfangszustand des Objekts mit dem minimalen Satz an Parametern, den es braucht, um zu existieren. Lass uns die Klasse **Cookie** modifizieren, so dass sie Parameter in ihrem Konstruktor akzeptiert.

```
class Cookie:
 # Constructor
 def __init__(self, name, shape, chips='Chocolate'):
  # Instance attributes
  self.name = name
  self.shape = shape
  self.chips = chips
```

In der **Cookie** Klasse muss jeder Cookie einen Namen, eine Form und Chips haben. Letzteres haben wir als „Chocolate“ definiert.

Auf der anderen Seite bezieht sich **self** auf die Instanz der Klasse (das Objekt selbst).

Versuche die Klasse in die Shell einzufügen und erstelle wie gewohnt eine Instanz des Cookies.

```
cookie2 = Cookie()
# TypeError
```

Du wirst einen Fehler erhalten. Das liegt daran, dass du den minimalen Satz an Daten angeben musst, den das Objekt zum Leben braucht – in diesem Fall **Name** und **Form**, da wir bereits **Chips** auf „Chocolate“ gesetzt haben.

```
cookie2 = Cookie('Awesome cookie', 'Star')
```

Um auf die Attribute einer Instanz zuzugreifen, musst du die Punktschreibweise verwenden.

```
cookie2.name
# 'Awesome cookie'
cookie2.shape
# 'Star'
cookie2.chips
# 'Chocolate'
```

Für den Moment hat die **Cookie** Klasse nichts allzu pikantes. Lass uns eine Beispielmethode **bake()** hinzufügen, um die Dinge interessanter zu machen.

```
class Cookie:
 # Constructor
 def __init__(self, name, shape, chips='Chocolate'):
  # Instance attributes
  self.name = name
  self.shape = shape
  self.chips = chips

 # The object is passing itself as a parameter
 def bake(self):
  print(f'This {self.name}, is being baked with the shape {self.sl
  print('Enjoy your cookie!')
```

Um eine Methode aufzurufen, verwende die Punktschreibweise und rufe es als Funktion auf.

```
cookie3 = Cookie('Baked cookie', 'Tree')
cookie3.bake()
# This Baked cookie, is being baked with the shape Tree and chips
Enjoy your cookie!
```

Die 4 Säulen der OOP in Python

Objektorientierte Programmierung beinhaltet vier Hauptsäulen:

1. Abstraktion

Die Abstraktion verbirgt die interne Funktionalität einer Anwendung vor dem Benutzer. Der Benutzer kann entweder der Endkunde oder andere Entwickler sein.

Wir können **Abstraktion** in unserem täglichen Leben finden. Du weißt zum Beispiel, wie du dein Telefon benutzt, aber du weißt wahrscheinlich nicht genau, was in ihm passiert, wenn du eine App öffnest.

Ein weiteres Beispiel ist Python selbst. Du weißt, wie man es benutzt, um funktionale Software zu erstellen, und du kannst es auch tun, wenn du das Innenleben von Python nicht verstehst.

Das Gleiche auf Code anzuwenden, erlaubt es dir, alle Objekte eines Problems zu sammeln und Standardfunktionalität in Klassen zu **abstrahieren**.

2. Vererbung

Vererbung erlaubt uns, mehrere **Unterklassen** von einer bereits definierten Klasse zu definieren.

Sie dient in erster Linie dazu, dem DRY-Prinzip zu folgen. Du wirst in der Lage sein, eine Menge Code wiederzuverwenden, indem du alle gemeinsam genutzten Komponenten in **Oberklassen** implementierst.

Du kannst es dir wie das reale Konzept der **genetischen Vererbung** vorstellen. Kinder (Subclass) sind das Ergebnis der Vererbung zwischen zwei Eltern (Superclasses). Sie erben alle physischen Eigenschaften (Attribute) und einige gemeinsame Verhaltensweisen (Methoden).

3. Polymorphismus

Polymorphismus erlaubt es uns, Methoden und Attribute der **Unterklassen**, die zuvor in der **Oberklasse** definiert wurden, leicht zu verändern.

Die wörtliche Bedeutung ist „**viele Formen**„. Das liegt daran, dass wir Methoden mit gleichem Namen, aber unterschiedlicher Funktionalität bauen.

Um auf die vorherige Idee zurückzukommen, sind Kinder auch ein perfektes Beispiel für Polymorphismus. Sie können ein definiertes Verhalten **get_hungry()** erben, aber auf eine etwas andere Art und Weise, zum Beispiel, dass sie alle 4 statt alle 6 Stunden hungrig werden.

4. Verkapselung

Verkapselung ist der Prozess, in dem wir die interne Integrität von Daten in einer Klasse schützen.

Obwohl es in Python keine **private** Anweisung gibt, kannst du Kapselung anwenden, indem du Mangling in Python benutzt. Es gibt spezielle Methoden, die **Getter** und **Setter** genannt werden, die es uns erlauben, auf einzigartige Attribute und Methoden zuzugreifen.

Stellen wir uns eine Klasse **Human** vor, die ein einzigartiges Attribut namens **_height** hat. Du kannst dieses Attribut nur innerhalb bestimmter Beschränkungen verändern (es ist fast unmöglich, höher als 3 Meter zu sein).

Einen Area Shape Resolver Calculator bauen

Eines der besten Dinge an Python ist, dass es uns erlaubt, eine große Vielfalt an Software zu erstellen, von einem CLI (Kommandozeileninterface) Programm bis hin zu einer komplexen Web-App.

Jetzt, wo du die Grundkonzepte von OOP gelernt hast, ist es an der Zeit, sie auf ein echtes Projekt anzuwenden.

Hinweis: Der gesamte folgende Code wird in diesem GitHub Repository verfügbar sein. Ein Code Revision Tool, das uns hilft, Codeversionen mit Git zu verwalten.

Deine Aufgabe ist es, einen Flächenrechner mit den folgenden Formen zu erstellen:

- Quadrat
- Rechteck
- Dreieck

- Kreis
- Sechseck

Shape Base Klasse

Erstelle zunächst eine Datei **calculator.py** und öffne sie. Da wir bereits die Objekte haben, mit denen wir arbeiten können, wird es einfach sein, sie in einer Klasse zu **abstrahieren**.

Du kannst die gemeinsamen Eigenschaften analysieren und herausfinden, dass es sich bei allen um **2D-Formen** handelt. Daher ist die beste Option, eine Klasse **Shape** mit einer Methode **get_area()** zu erstellen, von der jede Form erben wird.

Hinweis: Alle Methoden sollten Verben sein. Das liegt daran, dass diese Methode **get_area()** heißt und nicht **area()**.

```
class Shape:
 def __init__(self):
  pass

 def get_area(self):
  pass
```

Der obige Code definiert die Klasse; allerdings ist darin noch nichts Interessantes enthalten.

Lass uns die Standardfunktionalität der meisten dieser Shapes implementieren.

```
class Shape:
 def __init__(self, side1, side2):
```

```
        self.side1 = side1
        self.side2 = side2

    def get_area(self):
        return self.side1 * self.side2

    def __str__(self):
        return f'The area of this {self.__class__.__name__} is: {self.ge
```

Lass uns aufschlüsseln, was wir mit diesem Code machen:

- In der __**init**__ Methode fragen wir zwei Parameter ab, **side1** und **side2**. Diese bleiben als **Instanzattribute**
- Die Funktion **get_area()** gibt die Fläche der Form zurück. In diesem Fall benutzt sie die Formel für die Fläche eines Rechtecks, da es einfacher ist, sie mit anderen Formen zu implementieren.
- Die __**str**__**()** Methode ist eine „magische Methode“, genau wie __**init**__**()**. Es erlaubt dir, die Art und Weise, wie eine Instanz gedruckt wird, zu verändern.
- Das __**class**__**.**__**name**__ versteckte Attribut bezieht sich auf den Namen der Klasse. Wenn du mit einer **Triangle** Klasse arbeiten würdest, wäre dieses Attribut „Triangle“.

Rechteck Klasse

Da wir die Formel für den Flächeninhalt des Rechtecks implementiert haben, können wir eine einfache **Rechteck** Klasse erstellen, die nichts anderes tut, als von der **Shape** Klasse zu erben.

Um **Vererbung** in Python anzuwenden, erstellst du wie gewohnt eine Klasse und umschließt die Oberklasse, von der du erben möchtest, mit Klammern.

```
# Folded base class
class Shape: ...

class Rectangle(Shape): # Superclass in Parenthesis
 pass
```

Klasse Quadrat

Mit der Klasse **Quadrat** können wir einen ausgezeichneten Ansatz für **Polymorphismus** wählen.

Erinnere dich daran, dass ein Quadrat einfach ein Rechteck ist, dessen vier Seiten alle gleich sind. Das bedeutet, dass wir die gleiche Formel benutzen können, um den Flächeninhalt zu erhalten.

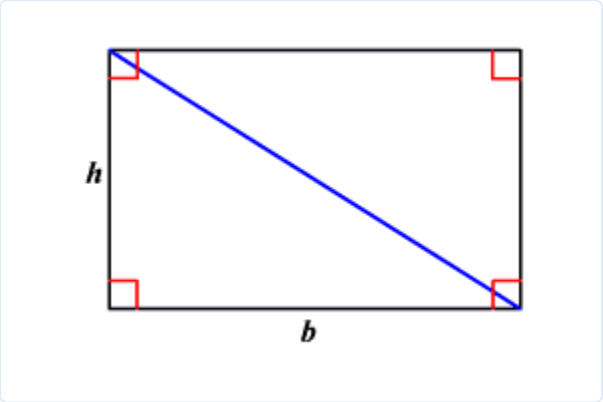
Wir können dies tun, indem wir die **init** Methode modifizieren, nur eine **Seite** als Parameter akzeptieren und diesen Seitenwert an den Konstruktor der **Rechteck** Klasse übergeben.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...

class Square(Rectangle):
 def __init__(self, side):
  super().__init__(side, side)
```

Wie du sehen kannst, übergibt die Superfunktion den Parameter **side** zweimal an die **Superklasse**. Mit anderen Worten, sie übergibt side sowohl als **side1** als auch als **side2** an den vorher definierten Konstruktor.

Dreiecks-Klasse

Ein Dreieck ist halb so groß wie das Rechteck, das es umgibt.



— Beziehung zwischen Dreiecken und Rechtecken (Bildquelle: Varsity tutors).

Daher können wir von der **Rectangle** Klasse erben und die **get_area** Methode so anpassen, dass sie der Formel für die Dreiecksfläche entspricht, die die Hälfte der Basis multipliziert mit der Höhe ist.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...

class Triangle(Rectangle):
 def __init__(self, base, height):
  super().__init__(base, height)

 def get_area(self):
```

```
    area = super().get_area()
    return area / 2
```

Ein weiterer Anwendungsfall der **super()**-Funktion ist es, eine in der **Superklasse** definierte Methode aufzurufen und das Ergebnis als Variable zu speichern. Das ist es, was in der **get_area()** Methode passiert.

Kreis Klasse

Du kannst die Kreisfläche mit der Formel **?r²** finden, wobei **r** der Radius des Kreises ist. Das bedeutet, dass wir die **get_area()** Methode modifizieren müssen, um diese Formel zu implementieren.

Hinweis: Wir können den ungefähren Wert von **?** aus dem Mathe-Modul importieren.

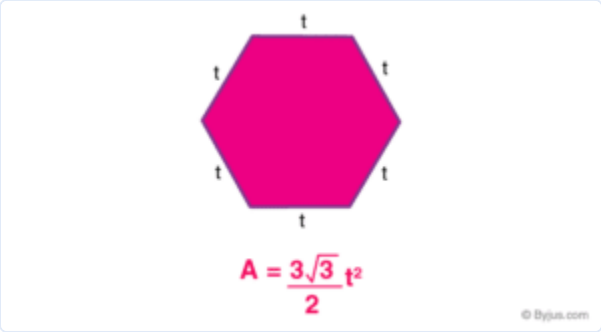
```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …

# At the start of the file
from math import pi

class Circle(Shape):
 def __init__(self, radius):
  self.radius = radius

 def get_area(self):
  return pi * (self.radius ** 2)
```

Der obige Code definiert die **Circle** Klasse, die einen anderen Konstruktor und **get_area()** Methoden verwendet.

Obwohl **Circle** von der Klasse **Shape** erbt, kannst du jede einzelne Methode umdefinieren und es nach deinen Wünschen auslegen.

Regelmäßige Hexagon Klasse

Wir brauchen nur die Länge einer Seite eines regulären Sechsecks, um seinen Flächeninhalt zu berechnen. Es ist ähnlich wie bei der Klasse **Square**, wo wir nur ein Argument an den Konstruktor übergeben.



— Sechseckige Flächenformel (Bildquelle: BYJU'S)

Allerdings ist die Formel ganz anders und es impliziert die Verwendung einer Quadratwurzel. Deshalb verwendest du die Funktion **sqrt()** aus dem Mathe-Modul.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
```

```
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …

# Import square root
from math import sqrt

class Hexagon(Rectangle):

 def get_area(self):
  return (3 * sqrt(3) * self.side1 ** 2) / 2
```

Testen unserer Klassen

Wenn du eine Python-Datei mit einem Debugger ausführst, kannst du in einen interaktiven Modus wechseln. Der einfachste Weg, dies zu tun, ist die Verwendung der eingebauten Breakpoint-Funktion.

Hinweis: Diese Funktion ist nur in Python 3.7 oder neuer verfügbar.

```
from math import pi, sqrt
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …
class Hexagon(Rectangle): …

breakpoint()
```

Starte nun die Python-Datei und spiele mit den erstellten Klassen herum.

```
$ python calculator.py

(Pdb) rec = Rectangle(1, 2)(Pdb) print(rec)
The area of this Rectangle is: 2
(Pdb) sqr = Square(4)
(Pdb) print(sqr)
The area of this Square is: 16
(Pdb) tri = Triangle(2, 3)
(Pdb) print(tri)
The area of this Triangle is: 3.0
(Pdb) cir = Circle(4)
(Pdb) print(cir)
The area of this Circle is: 50.26548245743669
(Pdb) hex = Hexagon(3)
(Pdb) print(hex)
The area of this Hexagon is: 23.382685902179844
```

Herausforderung

Erstelle eine Klasse mit einem **Methodenlauf**, in dem der Benutzer eine Form auswählen und deren Fläche berechnen kann.

Wenn du die Challenge abgeschlossen hast, kannst du einen Pull Request an das GitHub Repo senden oder deine Lösung im Kommentarbereich veröffentlichen.

Zusammenfassung

Objektorientierte Programmierung ist ein Paradigma, in dem wir Probleme lösen, indem wir sie als **Objekte** betrachten. Wenn du Python OOP verstehst, kannst du es auch leicht in Sprachen wie Java, PHP, Javascript und C# anwenden.

In diesem Artikel hast du gelernt:

- Das Konzept der Objektorientierung in Python

- Vorteile der objektorientierten gegenüber der strukturierten Programmierung
- Grundlagen der objektorientierten Programmierung in Python
- Konzept der **Klassen** und wie man sie in Python verwendet
- Der **Konstruktor** einer Klasse in Python
- **Methoden** und **Attribute** in Python
- Die vier Säulen der OOP
- Die Implementierung von **Abstraktion**, **Vererbung** und **Polymorphismus** in einem Projekt

Jetzt liegt es an dir!

Wenn dir dieser Leitfaden gefallen hat, schau dir unseren Beitrag über Python-Tutorials an.

Lass uns deine Lösung für die Herausforderung unten in den Kommentaren wissen! Und vergiss nicht, dir unseren Vergleich zwischen Python und PHP anzuschauen.