Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Das „Zen of Python“ klingt im ersten Moment nicht wie etwas, dass man intuitiv mit einer Programmiersprache verbinden würde, bietet aber einen geeigneten Einstieg in das Thema „Programmieren mit Python“ (Klein, 2021, S. 4). Die damit vermittelte Botschaft sollte mit einer Prise Humor gesehen werden, beschreibt im Kern jedoch die Gründe, warum Python zu so einer beliebten und weitverbreiteten Programmiersprache geworden ist. Softwareentwickler Tim Peters veröffentlichte 1999 das „Zen of Python“ mit dem Ziel, eine generelle Übersicht für den richtigen Umgang mit der Programmiersprache zu kreieren und neuen Python-Programmierern einige wichtige Grundsätze mit an die Hand zu geben, mit welchen sie das volle Potential der Sprache ausschöpfen können (Klein, 2021, S. 4). Kurz zusammengefasst ist die Botschaft des „Zen of Python“: Python wurde „(...) mit dem Ziel größter Einfachheit und Übersichtlichkeit entworfen. Zentrales Ziel bei der Entwicklung der Sprache ist die Förderung eines gut lesbaren, knappen Programmierstils.“ (Steyer, 2018, S. 3). Dies spiegelt sich unter anderem in der Syntax wieder, welche im Vergleich zu anderen Programmiersprachen näher an das Konzept der geschriebenen englischen Sprache angelehnt ist. Ergänzt wird die abstrakte Syntax unter Anderem um eine vielseitige, integrierte Standardbibliothek und die zur Verfügung stehenden Pakete im Python Package Index, welche von anderen Entwicklern frei zur Verfügung gestellt werden. So wird beispielsweise Software, welche von der Python-Community entwickelt und geteilt wird, leicht für andere Softwareentwickler zugänglich gemacht (PyPI, 2024). Neben einigen weiteren Aspekten spielen sogenannte Programmierparadigmen eine wichtige Rolle bei der Umsetzung eines lesbaren und knappen Programmierstils (Kastens, 2019). Sie üben großen Einfluss auf die Möglichkeiten, den fortführenden Umgang und die verschiedenen Vorgehensweisen bei der Nutzung einer Programmiersprache. Außerdem prägen die verschiedenen Paradigmen zahlreiche weitere Prozesse in der Funktionsweise einer Programmiersprache, wie beispielweise dem Umgang mit Daten, Befehlen und dem allgemeinem Stil, in welchem ein Programm letztendlich auch geschrieben werden sollte, um es so leistungsfähig wie möglich zu gestalten. Deshalb wird im Folgenden zunächst einmal ein Blick darauf geworfen, wie Paradigmen allgemein funktionieren, um so perspektivisch die Grundlage für eine genauere Betrachtung von Python zu bilden. Ein besonderer Fokus soll dabei auf das objektorientierte Programmierparadigma gerichtet werden.

Ziel der Arbeit ist es, die wichtigsten Konzepte der objektorientierten Programmierung in Python erst theoretisch, anschließend mit einem Beispielprogramm zu erläutern, um einen Überblick darüber zu bekommen, welche Möglichkeiten dieses besondere Programmierparadigma mit sich bringt. Dabei sollen Stärken, aber auch mögliche Schwachpunkte der objektorientierten Programmierung hervorgehoben werden.

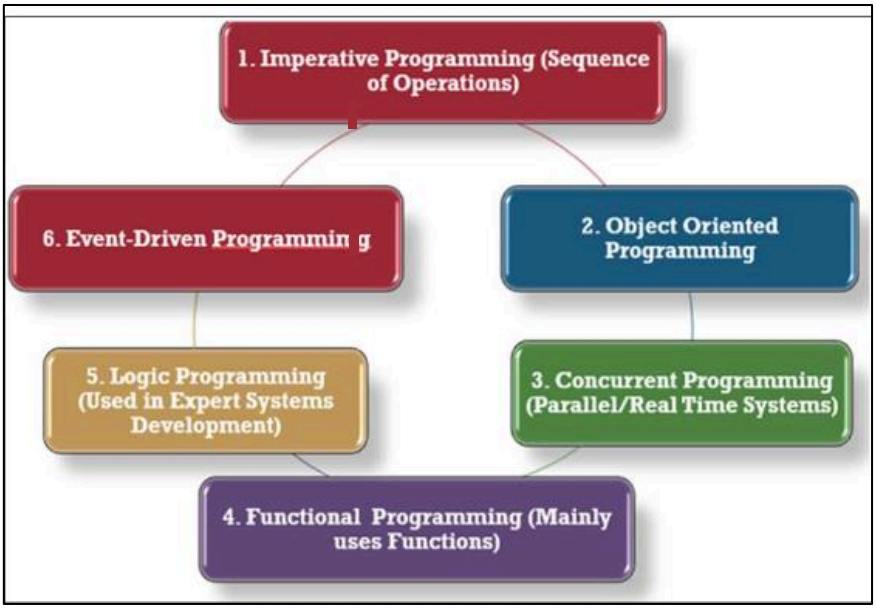
2. Programmierparadigmen

2.1 Theoretische Grundlagen eines Programmierparadigmas

Um sich genauer mit dem Thema auseinandersetzen zu können, lohnt es sich zunächst einmal ein Verständnis dafür zu erlangen, was genau denn ein Programmierparadigma ist und inwiefern es letztendlich in Python entwickelte Software beeinflusst. Anschließend wird der Fokus auf die objektorientierte Programmierung gerichtet, das Programmierparadigma, welches in Python neben weiteren wichtigen Funktionen eine problemlösungsorientierte Vorgehensweise seitens des Entwicklers ermöglicht. Programmierparadigmen, welche im Grunde eine Anzahl an unterschiedlichen Prinzipien bezeichnen, die die Ausführung von Programmen beschreiben, spielen in Python, aber auch anderen Programmiersprachen, ebenfalls eine wichtige Rolle bei der Erreichung eines lesbaren und gut strukturierten Programmierstils (Kastens, 2019). In Abbildung 1 unterhalb werden die verschiedenen Programmierparadigmen zunächst dargestellt, um zu zeigen, dass es verschiedene Konzepte gibt, mit denen wiederum unterschiedliche Ziele in einer Vielzahl von Programmiersprachen verfolgt werden.

Abbildung 1

Die verschiedenen Programmierparadigmen



Anmerkung: (Monday et al., 2021, S. 28)

Die Programmiersprache selbst, welche sich in der Basis aus verschiedenen Symbolen und einem bestimmten Schreibstil zusammensetzt, bietet zunächst einmal die Grundlage dafür, dass verschiedene Programmierparadigmen überhaupt existieren und eingesetzt werden können (Brborich et al., 2020).

Der Begriff Programmierparadigma beschreibt dabei viel allgemeiner den Stil beziehungsweise die zugrundeliegenden Konzepte und Methoden, die einem Programmierer zur Verfügung stehen, um Software in einer bestimmten Programmiersprache zu schreiben sowie zu pflegen (Brborich et al., 2020). Dennoch steht eine Programmiersprache in enger Wechselwirkung mit ihrem Paradigma bzw. Paradigmen. Um das Konzept greifbarer zu machen, lohnt es sich, einen Blick auf die Anfänge der Computertechnik zu werfen: Während man anfangs noch gezwungen war, Programme in einer für Maschinen direkt verständlichen Sprache zu verfassen, ist man inzwischen nicht mehr an die Nutzung der Maschinensprache gebunden (Neumann, 2020, S. 9). Durch Compiler bzw. Interpreter wird die menschliche Sprache in für Maschinen verständliche Sprache übersetzt, wodurch es für den Programmierer nicht mehr nötig ist, konkrete Anweisungen in Maschinensprache zu verfassen, um dem Prozessor exakt mitzuteilen, welche Anweisung wann und wie ausgeführt werden soll (Neumann, 2020, S. 10). Python ist eine interpretierte Sprache, bedeutet also, dass ein Programm während der Ausführung Zeile für Zeile in Maschinensprache übersetzt wird, wobei im Gegensatz dazu beim Compiler zunächst der ganze Code übersetzt und anschließend ausgeführt wird. Trotz der strukturellen Unterschiede beider Prozesse ermöglicht ein Interpreter bzw. Compiler ein höheres Maß an Abstraktion, und ermöglicht durch eine weniger direkte Kommunikation mit der Hardware einen leichteren Einstieg in die Nutzung der Programmiersprache. So können verschiedene Bausteine in der Software intuitiver und mit einem größerem Maß an Freiheit entwickelt werden (Neumann, 2020, S. 11). Diese Flexibilität geht mit der Notwendigkeit für gewisse Rahmenbedingungen einher, die bei all den Freiheiten, die eine Programmiersprache wie Python bietet, trotzdem gewisse Vorgaben und Prinzipien bilden, anhand welcher sich Softwareentwickler orientieren können (Brborich et al., 2020). Und eben diese angesprochenen „Rahmenbedingungen“ bilden Programmierparadigmen. Sie dienen dem Programmierer wie eine Art Leitfaden, mithilfe dessen geordnete und logische Software geschrieben werden kann, welche dann von einem Computer verarbeitet wird. Dieser Leitfaden beinhaltet, je nach Paradigma, vorgegebene Muster und Richtlinien, welche abhängig von dem zu lösenden Problem unterschiedliche Herangehensweisen ermöglichen und deshalb Flexibilität bei der Lösung unterschiedlichster Probleme bieten.

2.2 Das objektorientierte Programmierparadigma

2.2.1 Objekte

Die Idee, möglichst kurze und voneinander abgekapselte Programme zu schreiben, die dann wiederum in ihrer Gesamtheit in der Lage sind, komplexe Systeme abzubilden, bildet bei der objektorientierten Programmierung die Grundlage. Erreicht werden soll dies unter anderem durch ein hohes Maß an Abstraktion, wodurch reale Abläufe möglichst realitätsnah in einem Programm dargestellt werden sollen. Wie wird die angesprochene Realitätsnähe erreicht? Zunächst einmal über die Einführung von sogenannten Objekten. Im Gegensatz zu realen Objekten, mit welchen man haptisch und auch funktionell interagieren kann, man denke beispielsweise an ein Auto, bei welchen die Betätigung der Warnsignaltaste die Warnblinker aktiviert oder das Drücken der Lenkradmitte die Hupe auslöst, ist es bei der Entwicklung von Software nicht möglich, das Objekt mit den eigenen Händen zu berühren. Nichtsdestotrotz ist der Grundgedanke nicht weit von dem entfernt, was wir intuitiv mit einem Objekt verbinden. Objekte in Programmen sind Modelle, die eine gewisse Aufgabe besitzen und mit denen auf verschiedene Weise interagiert werden kann (Phillips, 2018, S. 7). In diesem Zusammenhang sind Objekte eine Ansammlung von Daten und den dazugehörigen Zuständen, Funktionen und Verhaltensweisen, welche die Aufgabe besitzen, reale oder abstrakte Gegenstände abzubilden (Phillips, 2018, S. 8). Sowohl in der objektorientierten Programmierung als auch in der realen Welt bezieht sich ein objektorientierter Lösungsansatz also darauf, die Komplexität von Problemen durch Abstraktion von Wissen und Abkapselung der einzelnen Objekte zu bewältigen (Lee, 2019, S.92). Ein Beispiel dafür wäre das erfolgreiche Managen eines Flughafens: Zahlreiche kleinere Vorgänge, wie die Verwaltung des Gepäcks, die Koordination der ankommenden und abreisenden Flugzeuge, die Sicherheitskontrollen, oder auch die Verwaltung von Passagierinformationen. Jeder einzelne dieser Prozesse geht mit eigenen Herausforderungen einher, die unterschiedliche Lösungen erfordern. Dennoch kann das Gesamtkonstrukt Flughafen nur erfolgreich weiterlaufen, wenn all diese „kleineren“ Prozesse perfekt ineinander greifen. Um dies zu ermöglichen, werden in der Regel alle Aufgaben in die Hände von Teams gelegt, die explizit eine dieser Aufgaben bearbeiten und darauf spezialisiert sind. Diese Teams haben dann wiederum ihre eigenen Methoden und Mittel zur Verfügung, um die Aufgaben effektiv zu erledigen. Man denke dabei beispielweise an die Fluglotsen im Air Traffic Control Tower, die alle Starts, Landungen und den Verkehr auf dem Boden koordinieren. Nicht einer dieser Tower ist zuständig für den weltweiten Flugverkehr, sondern jeder Tower hat einen bestimmten Radius in welchem er für eine gewisse Anzahl an Flugzeugen zuständig ist. Innerhalb des Radius wird die Gesamtzahl der verkehrenden Flugzeuge wiederum auf zahlreiche Fluglotsen verteilt. Auch der Aufgabenbereich der einzelnen Fluglotsen wird dann wieder auf weitere Objekte zergliedert, beispielsweise Flugzeuge oder auch bestimmte Flugrouten. Und wie bereits am Anfang erwähnt, jedes Objekt verfügt dabei über eigene Funktionen, Aufgaben und Spezialisierungen.

Diese modulare Zergliederung in einzelne, abstrahierte Verantwortungsbereiche bildet auch das Grundkonzept der objektorientierten Programmierung. Modularität hängt im Kontext des objektorientierten Programmierparadigmas auch mit dem Thema Wiederverwendbarkeit und Abstraktion zusammen. Wie genau dies erreicht wird, wird nun im nächsten Abschnitt genauer untersucht.

2.2.2 Klassen

Für den Zweck dieser Arbeit wird nun tiefer auf das Flughafenbeispiel aus dem vorherigen Kapitel eingegangen. Obwohl das Objekt „Flugzeug“ aus der Perspektive des Fluglotsen einige eindeutige Merkmale wie die spezifische Flugnummer besitzt, gibt es doch einige Gemeinsamkeiten zu all den weiteren Flugzeugobjekten im näheren Radius. Eine Gemeinsamkeit ist eben die Tatsache, dass jedes davon eine Flugnummer besitzt. Des Weiteren gibt es verschiedene Flugzeugmodelle, diese können beispielsweise Passagiere befördern oder auch Güter, wobei verschiedene Modelle wiederum verschiedene Kapazitäten besitzen. Anders formuliert, obwohl jedes Objekt eindeutige Merkmale für sich besitzt, sind diese spezifischen Merkmale bei all den Flugzeugobjekten wiederkehrend. Um also präziser bei der Organisation des Flugverkehrs vorgehen zu können, kann es also durchaus von Vorteil sein, eine übergeordnete Vorlage für das Objekt Flugzeug anzulegen. Die Vorlage definiert anschließend mit welchen allgemeinen Eigenschaften ein solches Objekt einhergehen kann. Und genau diese Aufgabe besitzen Klassen in der objektorientierten Programmierung. Klassen ermöglichen es, reale Objekte in einem Programm abzubilden und diese mit eigenen, wiederkehrenden Merkmalen zu versehen.

Objekte werden dann auf Basis der vorher definierten Klasse samt ihrer möglichen Eigenschaften und Funktionen definiert. Somit bildet eine Klasse die Grundlage für jedes Objekt und ermöglicht eine klare Strukturierung, da mögliche Merkmale und Eigenschaften eines Objekts nur einmal in Form einer übergeordneten Klasse definiert werden, anschließend aber beliebig viele Objekte anhand dieser Blaupause erstellt werden können. Im Beispiel des Flughafens würde das bedeuten, anstatt für jedes einzelne Flugzeugobjekt die Details neu definieren zu müssen, definiert man alle möglichen Flugzeugmerkmale in einer Klasse vor, so wie beispielsweise Flugzeugmodell und Flugnummer, und füllt dann anschließend für jedes einzelne Objekt nur die fehlenden Informationen beim Erschaffen eines Objektes ein.

Im weiteren Verlauf der Arbeit wird das Konzept der objektorientierten Programmierung in Form eines Programms für das Flughafenbeispiel veranschaulicht. Das Beispielprogramm wurde in Python erstellt. In der Abbildung 2 erkennt man im ersten Schritt des Programms, wie Klassen als Blaupausen für Objekte dienen und inwiefern die beiden Bausteine miteinander interagieren.

Abbildung 2

Klasse Flugzeug

```
class Flugzeug:
    def __init__(self, model, flight_number, status):
        self.model = model
        self.status = status
        self.flight_number = flight_number


    def __str__(self):
        return f"Flugzeugmodell: {self.model}, Flugnummer: {self.flight_number}, Status des Flugzeugs: {self.status}"
```

Im ersten Schritt wird die Klasse „Flugzeug“ mithilfe der Syntax „class Flugzeug:“ benannt und definiert. Danach kommt der __init__ Konstruktor, welcher für die Erschaffung eines neuen Objektes zuständig ist und dieses mit den festgelegten Attributen (model, status, flight_number) versieht, jedes Mal wenn die Klasse genutzt wird, um ein Objekt zu kreieren. Die Attribute (self, model, flight_number, status) werden hier definiert, ohne dabei selber konkrete Informationen zu enthalten. Die Erschaffung, auch bekannt als Instanziierung eines Objektes, soll dem Nutzer hier bestätigt werden. Deshalb wird mit der Funktion „def __str__“ festgelegt, in welcher Form das Objekt dem Nutzer nach der Erschaffung in der Ausgabe angezeigt werden soll.

Durch den Parameter „self“ erkennt Python stets, um welches Objekt es sich gerade handelt, da komplexere Programme in der objektorientierten Programmierung zum Teil zahlreiche verschiedene Objekte enthalten können. Die Funktion „__str__“ und der Parameter „self“ werden später nochmals genauer betrachtet. Hier soll zunächst die Erschaffung eines Objektes im Vordergrund stehen.

Die Instanziierung selbst erfolgt in den Zeilen 12 bis 13, hervorgehoben in der unteren Abbildung 3.

Abbildung 3

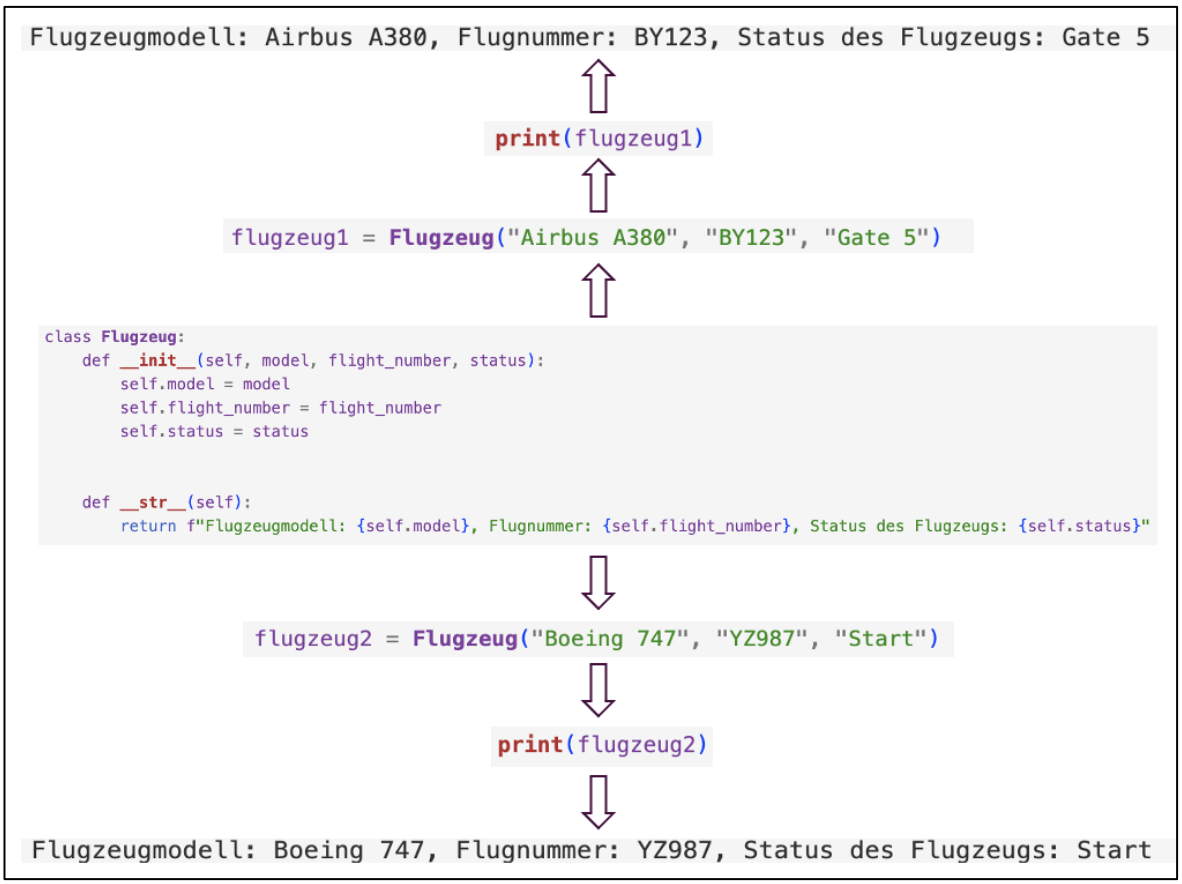
Instanziierung von zwei Flugzeugobjekten anhand der Klasse „Flugzeug“

```
12 flugzeug1 = Flugzeug("Airbus A380", "BY123", "Gate 5")
13 flugzeug2 = Flugzeug("Boeing 747", "YZ987", "Start")
14 print(flugzeug1)
15 print(flugzeug2)
```

Ein Objekt wird auf Grundlage einer definierten Klasse kreiert, indem man den Namen der Klasse, in diesem Fall also „Flugzeug“, mithilfe der Parameter (model, flight_number, status) in das Programm implementiert. Hier werden die beiden Objekte zu Demonstrationszwecken in den Variablen „flugzeug1“ und „flugzeug2“ gespeichert und anschließend mithilfe der „print()-Funktion“ auf den Bildschirm des Nutzers gedruckt. Betrachtet man anschließend das Resultat der beiden „print()-Funktionen“ im Terminal, demonstriert dies eine der großen Stärken der objektorientierten Programmierung in Python.

Abbildung 4

Prozess der Objektinstanziierung anhand der Klasse „Flugzeug“



Auf Basis der Klasse „Flugzeug“ werden in unserem Beispiel zwei, grundsätzlich aber beliebig viele, einzelne Flugzeugobjekte mit unterschiedlichen Eigenschaften und Zuständen erschaffen. Abbildung 4 oberhalb soll den Prozess der Objektinstanziierung graphisch veranschaulichen. Aus der Klasse Flugzeug entspringen hier zwei individuelle Objekte mit unterschiedlichen Attributsdaten, erkennbar unter Anderem anhand der unterschiedlichen Flugzeugmodelle. Das eine Objekt, gespeichert in der Variablen „flugzeug1“, hat beispielsweise ein anderes Flugzeugmodell als das zweite Objekt „flugzeug2“. Erkennbar ist dies nach der Ausführung von „print(flugzeug1)“ und „print(flugzeug2)“, sobald die erschaffenen Objekte als Resultat in der Ausgabe angezeigt werden. Im Kontext der Programmierung verhindert diese Vorgehensweise die Duplikation von Code-Blöcken, da nur einmalig in der Klasse „Flugzeug“ die Attribute definiert werden. Außerdem strukturiert es das Programm in einer klaren Weise und erleichtert die Implementierung von weiteren Funktionalitäten, selbst bei komplexen Programmen, da man hier theoretisch zahlreiche weitere Flugzeuge in das Programm aufnehmen könnte. Ein weiteres, wichtiges Grundkonzept der objektorientierten Programmierung ist die dynamische Veränderung und Überschreibung von Daten innerhalb der Objekte. Dies geschieht mithilfe von sogenannten Methoden, welche im nächsten Kapitel genauer betrachtet werden.

2.2.3 Methoden

Klassen fungieren wie eine Art vordefinierte Blaupause zur Erstellung von Objekten und ermöglichen die realitätsnahe Beschreibung von Eigenschaften in der objektorientierten Programmierung. Neben der Dokumentation von Merkmalen wie beispielsweise zum Typ des Flugzeugs bzw. der Flugnummer helfen Klassen aber auch dabei, Funktionalitäten eines Objektes darzustellen. Dies geschieht mithilfe von sogenannten Methoden. Ist es klar, dass Objekte in einem Softwaremodell eine gewisse Auswahl an möglichen Verhaltensweisen und Funktionalitäten besitzen, so kann dies innerhalb einer Klasse in Form einer Funktion definiert werden und eine solche Funktion wird als Methode bezeichnet (Phillips, 2018, S. 13). Für jedes individuelle Objekt kann so das Verhalten durch das Aufrufen der Methode samt möglicher Funktionsparameter beeinflusst werden. Die Funktionsweise einer Methode soll nun mit Hilfe eines Beispiels genauer betrachtet werden: Im Kontext des Flughafenbeispiels könnten Methoden zur Festlegung dessen genutzt werden, welche Aktivität ein Flugzeug gerade durchführt, also ob es sich beispielsweise gerade beim Starten, Landen oder parkend am Gate befindet.

Abbildung 5

Methode zur Aktualisierung des Status eines Flugzeuges

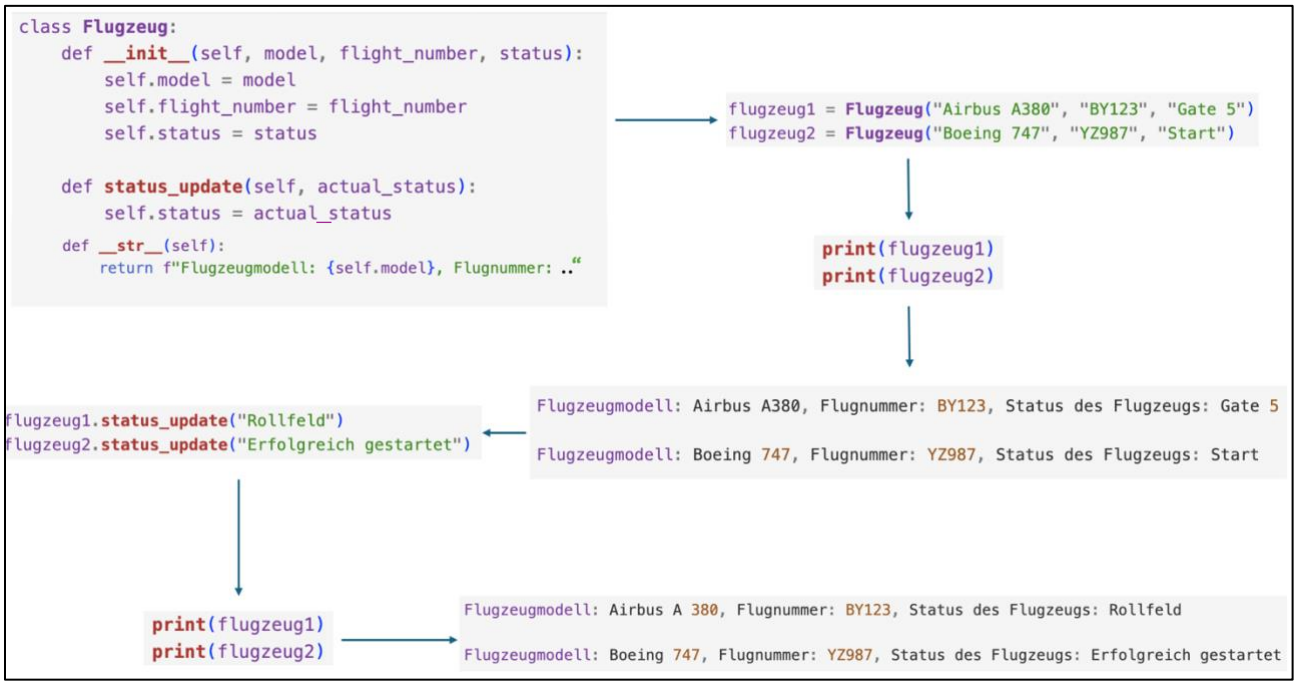
```
1  class Flugzeug:
2      def __init__(self, model, flight_number, status):
3          self.model = model
4          self.flight_number = flight_number
5          self.status = status
6
7
8      def __str__(self):
9          return f"Flugzeugmodell: {self.model}, Flugnummer: {self.flight_number}, Status des Flugzeugs: {self.status}"
10
11     def status_update(self, actual_status):
12         self.status = actual_status
```

In der obigen Abbildung 5 wird die Verwendung von Methoden zum Aktualisieren des Flugstatus dargestellt. Tatsächlich hat das vorherige Beispiel bereits eine Methode enthalten, nämlich die in Python vordefinierte Methode „__str__“ (Zeile 8, Abbildung 5). Wie man am vorherigen Beispiel erkennen kann, gibt die „__str__“-Methode dem Nutzer des Programms eine Nachricht in Textform zurück, nachdem ein Objekt erfolgreich erschaffen wurde und dieses mit einer „print()“-Funktion in der Ausgabe angezeigt werden soll. Dabei werden im Beispielprogramm zwei Flugzeugobjekte erschaffen (flugzeug1, flugzeug2, siehe Abbildung 4), welche beide in der Ausgabe auf Basis der „__str__“-Methode im selben Format angezeigt werden. In diesem Zusammenhang ist die Methode in der Zeile 11 aber Besonders interessant, in dem Beispiel genannt „status_update“, da sie es zur Aufgabe hat, den momentanen Zustand des Flugzeuges zu beschreiben und falls nötig, zu überschreiben.

Anhand der Syntax der Methode „status_update“ in Zeile 11 erkennt man, dass sich Methoden im Aufbau größtenteils nicht von normalen Funktionen außerhalb von Klassen unterscheiden. Ein Unterschied besteht jedoch bei der Verwendung des Parameters „self“ in der Methode, weshalb im Folgenden betrachtet werden soll, wie eine Methode die Attribute eines Objektes verändern kann und welche Rolle der Parameter „self“ dabei spielt. Abbildung 6 soll darstellen, wie schnell und direkt eine Überschreibung des Flugzeugstatus in der objektorientierten Programmierung erfolgen kann und auf welche Weise Methoden die Wiederverwendbarkeit und den Zugriff auf Informationen innerhalb eines Objektes abstrahieren:

Abbildung 6

Überschreibung des Attributes „model“



Die Methode „status_update“ in unserem Beispiel enthält neben dem Parameter „actual_status“ den Parameter „self“. Der Parameter „self“ lässt den Interpreter während der Ausführung des Programms wissen, dass die Methode zu allen Objekten gehört, die auf Basis der Klasse Flugzeug erschaffen werden (Phillips, 2018, S. 36). Folglich führt dies dazu, dass die Methode „status_update“ direkt auf die Eigenschaften aller Objekte zugreifen kann, ohne dass konkrete Daten zu den Objekten an die Methode übermittelt werden müssen (Phillips, 2018, S. 37). Es reicht lediglich der Name des spezifischen Objektes für den Zugriff (bspw. „**flugzeug1**.status_update(„Rollfeld“)“, Abbildung 6). Da jede Methode innerhalb einer Klasse den Parameter „self“ als erstes Argument enthält, kann die Vorgehensweise auch für alle Attribute im Objekt eingesetzt werden. So wie im Flughafenbeispiel der Zustand „Status“ verändert wird, könnten alle weiteren möglichen Eigenschaften des Flugzeuges ebenso überschrieben werden, entweder einzeln oder auch in ihrer Gesamtheit.

Innerhalb der Methode „status_update“ wird hier lediglich der Status des Flugzeuges mit dem aktuellen Status überschrieben, welcher beim Aufrufen als Parameter hinzugefügt wird (self.status = actual_status, Abbildung 6). Die konkreten Informationen sind als mögliche Parameter in der Methode enthalten, in diesem Fall dem aktualisierten Status des Flugzeuges („Rollfeld“ bzw. „Erfolgreich gestartet“). Wie man also erkennen kann, können Attribute direkt und strukturiert überschrieben werden und zwar für alle Objekte, die auf Grundlage der Klasse Flugzeug instanziiert wurden, welche die Methode „status_update“ enthält.

Die Implementierung von Methoden bietet somit ein Mittel zur Beeinflussung des Verhaltens und hilft dabei, Zusammenhänge zwischen Objekten und Eigenschaften zu kontrollieren. Attribute können situationsbedingt gesteuert werden, um Prozesse möglichst strukturiert und dynamisch zu gestalten. Außerdem sind alle benötigten Informationen und deren mögliche Aktionen in einer Klasse zusammengefasst und werden nur an Objekte weitergegeben, die auf Basis dieser Klasse kreiert werden. So entstehen abgekapselte Einheiten innerhalb eines Programms, in welchen Klassen und die darauf aufbauenden Objekte zusammengefasst werden, ohne dabei für jedes Objekt logische Zusammenhänge neu definieren zu müssen. Diese Aufgabe übernehmen Klassen und damit zusammenhängend Methoden. Das Resultat ist die klare Trennung von Abläufen und die Möglichkeit, durch direkten Zugriff auf Attribute von Objekten zuzugreifen und diese zu ändern. Die syntaktische Struktur bleibt dabei stets nachvollziehbar, selbst bei komplexen Programmen mit vielen verschiedenen Komponenten. Die Idee der Kapselung von Objekten samt ihrer Verhaltensweisen und Attribute spielt noch aus einem weiteren Grund eine zentrale Rolle bei der objektorientierten Programmierung.

2.2.4 Kapselung in der objektorientierten Programmierung

Neben den strukturellen Vorteilen einer Aufteilung des Programms in einzelne prozessuale Code-Blöcke, um selbst komplexe Systeme übersichtlich und logisch zu gestalten, ist die Kapselung darüber hinaus ein wichtiges Mittel zum Schutz der Integrität der im Programm enthaltenen Informationen (Gabbrielli & Martini, 2023, S. 284). Die Idee, dass nur über spezifisch definierte Zugangspunkte sowie Methoden auf sensitive Daten zugegriffen werden kann, soll die versehentliche oder auch bewusste Manipulation von Informationen in einem objektorientierten Programm erschweren. Im Kontext des Flughafenbeispiels würde das im Konkreten bedeuten: Interne Daten, welche jedes Flugzeugobjekt besitzt, könnten bei der Entwicklung oder Wartung des gesamten Programms nur von Entwicklern bearbeitet werden, indem diese mithilfe einer Methode darauf zugreifen. Der Status des Flugzeuges könnte somit beispielsweise nur auf „Erfolgreich gestartet“ aktualisiert werden, wenn dafür die Methode „status_update“ aufgerufen wird, welche genau diese Aufgabe besitzt.

Um das zu erreichen gibt es in der objektorientierten Programmierung in Python einige Konventionen, welche dem Programmierer ein Mittel an die Hand geben, um über die Art und Weise zu entscheiden, wie mit Daten interagiert werden kann und bei welchen der Zugang erschwert werden soll (Steyer, 2018, S. 145). Durch verschiedene Richtlinien in der Syntax der objektorientierten Programmierung in Python soll dem Programmierer die Möglichkeit gegeben werden, verschiedene Sichtbarkeitsstufen für die Attribute eines Objektes vorzugeben (Steyer, 2018, S. 161). Grundsätzlich sind Attribute und Methoden in Python public, also öffentlich zugänglich für alle am Programm arbeitenden Entwickler, indem mithilfe einer Referenz direkt darauf zugegriffen und die Informationen somit auch verändert werden können (Steyer, 2018, S. 161). Wie man anhand des Outputs in Abbildung 7 unterhalb erkennt, kann durch Referenz auf das Objekt Flugzeug mithilfe von „flugzeug.model“ (Abbildung 7, Zeile 15) jegliche Information zum Flugzeugmodell im Programm aufgerufen oder überschrieben werden (Modell des Flugzeugs: „Airbus A-380“ wird überschrieben von „Boeing 747-400“).

Abbildung 7

Überschreibung des Attributes „model“

```
1  class Flugzeug:
2      def __init__(self, model, flight_number, status):
3          self.model = model #öffentlicher Zugang
4          self.flight_number = flight_number # öffentlicher Zugang
5          self.status = status # öffentlicher Zugang
6
7      def __str__(self):
8          # lesbare Textdarstellung des Objekts bei Verwendung der print()-Funktion
9          return f"{self.model}, {self.flight_number}, {self.status}"
10
11 flugzeug = Flugzeug("Airbus A-380", "AB123", "Gate 5") # Instanziierung des Flugzeugobjekts
12
13 print(flugzeug) # Anzeige des Flugzeugobjekts "flugzeug"
14             ↳ Output: Airbus A-380, AB123, Gate 5
15 flugzeug.model = "Boeing 747-400" # Änderung des Attributs "model"
16
17 print(flugzeug) # Anzeige des Flugzeugobjekts "flugzeug" mit geänderten Attributen
               ↳ Output: Boeing 747-400, AB123, Gate 5
```

Das Modell des Objektes „flugzeug“ ändert sich folglich ohne jegliche Zugangsbeschränkung und könnte nachgehend weiterhin beliebig bearbeitet werden und zwar von jedem, der über die Referenz „flugzeug.model“ darauf zugreift. Bei komplexeren Programmen muss aber darauf geachtet werden, dass bei der Entwicklung sowie Wartung der Software keine sensiblen Daten einfach überschrieben werden können. Um dies zu gewährleisten, gibt es in Python Mechanismen, welche dies kontrollieren sollen. Auf der einen Seite geschieht dies durch die Kapselung von Klassen und ihren Objekten, um durch den Aufbau des Programms darzustellen, welche Informationen zu welchen Prozessen gehören. Auf der anderen Seite gibt es Methoden, welche dafür gedacht sind, direkte Zugangspunkte zu den Daten innerhalb eines Objektes zu erschaffen.

Es gibt zusätzlich gewisse Richtlinien in der Syntax von Python, mit welchen Entwickler die Merkmale eines Objektes als privat kennzeichnen können, sodass Kontrolle darüber übernommen werden kann, wie auf Attribute zugegriffen wird (Steyer, 2018, S. 162). Dabei ist zu beachten, dass es in Python auch bei privat gekennzeichneten Attributen einen Weg gibt, auf diese zuzugreifen, weshalb es sich hierbei tatsächlich mehr um Konventionen handelt, als ein striktes Regelwerk. Um den Prozess der Zugangsbeschränkung in der objektorientierten Programmierung in Python zu verdeutlichen und aufzuzeigen, dass es sich dabei eher um Konventionen handelt als Regeln, wird im Folgenden dieselbe Klasse wie in Abbildung 7 mit privat gekennzeichneten Attributen versehen und dann versucht, darauf zuzugreifen. In Abbildung 8 soll verdeutlicht werden, dass zunächst einmal nur eine kleine Änderung in der Syntax bereits einen Unterschied in der Zugänglichkeit des Attributes „flugzeug.model“ bewirkt.

Abbildung 8

Zugriff auf privates Attribut des Objektes „flugzeug“

```
1  class Flugzeug:
2      def __init__(self, model, flight_number, status):
3          self.__model = model  # privates Attribut
4          self.__flight_number = flight_number  # privates Attribut
5          self.__status = status # privates Attribut
6
7  def __str__(self):
8          # lesbare Textdarstellung des Objekts bei Verwendung der print()-Funktion
9          return f"{self.model}, {self.flight_number}, {self.status}"
10
11 # Instanziierung des Flugzeugobjekts
12 flugzeug = Flugzeug("Airbus A380", "AB123", "Gate 5")
13
14 # Anzeige des Flugzeugobjekts "flugzeug"
15 print(flugzeug.__model)
```

Output:

```
    print(flugzeug.__model)
          ^^^^^^^^^^^^^^^^
AttributeError: 'Flugzeug' object has no attribute '__model'
```

Anhand der Klasse „Flugzeug“ kann in den Zeilen 1 bis 5 erkannt werden, wie die Syntax in Python für eine private Kennzeichnung eines Attributes aussieht. Indem man zwei Unterstriche vor dem Namen des Attributs hinzufügt (zum Beispiel Zeile 3: „self.__model = model“), wird Python mitgeteilt, dass es sich hier um private Daten handelt, die nicht einfach jederzeit zugänglich und veränderbar sein sollen. Die nächsten Zeilen des Programms in Abbildung 8 sind identisch zum vorherigen Beispiel mit öffentlich zugänglichen Attributen, in Zeile 15 wird jedoch der Unterschied deutlich: Nach der Instanziierung des Objektes „flugzeug“ in Zeile 12 kann das Modell des Flugzeuges nicht einfach aufgerufen werden, indem man den in der Klasse definierten Namen und das Objekt referenziert. Stattdessen bekommt man eine Fehlermeldung angezeigt, welche den Nutzer wissen lässt, dass ein bestimmtes Attribut mit dem Namen „__model“ nicht in der Klasse „Flugzeug“ definiert wurde.

Wie man aber anhand der Abbildung 8 erkennen kann, wurde das Attribut „model“ in der Klasse definiert, zwar unter Verwendung einer anderen Syntax („self.__model“), nichtsdestotrotz wird die Veränderung der Syntax in Zeile 15 in der „print(flugzeug.__model)“-Anweisung berücksichtigt und sollte deshalb aufrufbar sein. Weshalb kann das Attribut jedoch nicht gefunden werden? Grund für die Fehlermeldung ist ein Mechanismus, durch welchen ein Attribut in Python intern umbenannt wird, sobald es mit den beiden Unterstrichen als privat gekennzeichnet wird (Iona et al., 2023, S. 20). Im Konkreten erkennt Python beim Interpretieren des Programms die Syntax für eine private Kennzeichnung eines Attributes und benennt das Attribut intern zu „_Classname_Attribute“ um, in diesem Fall „_Flugzeug_model“ (Iona et al., 2023, S. 20).

Die interne Umbenennung des Attributs wird als Name Mangling bezeichnet, da Python den Namen des Attributes verändert, um den Zugriff nur über Umwege zu erlauben und die Zugänglichkeit von Daten durch Obskurität zu erschweren (Iona et al., 2023, S. 22). Dieser Schritt hat zur Folge, dass ein Zugriff über den in der Klasse definierten Attributsnamen mit der in Abbildung 8 gesehenen Fehlermeldung einhergeht. Kennt der Nutzer jedoch den geänderten Namen des Attributes, könnte er theoretisch weiterhin unter Referenz des geänderten Namens darauf zugreifen (zum Beispiel durch „print(flugzeug._Flugzeug__model)“), weshalb das Name Mangling in Python eine Konvention darstellt, welche bei der Entwicklung von Programmen eingehalten werden sollte, um die Datenintegrität zu gewährleisten. Hierin unterscheidet sich Python von einigen anderen Programmiersprachen, bei welchen der Zugang zu privaten Daten von außen durch strikte Mechanismen verhindert wird, wohingegen bei Python das Vertrauen in die Entwickler gelegt wird, weshalb die Datenintegrität eng mit der Einhaltung der Rahmenbedingungen verknüpft ist (Iona et al., 2023, S. 20). Zusammengefasst wird bei der objektorientierten Programmierung in Python also durch die Kombination aus der logischen Kapselung von Code-Blöcken und der Verwendung von Name Mangling ein gewisses Maß an Zugangskontrolle in Bezug auf sensible Daten erreicht. Die Einhaltung der Konventionen seitens der Entwickler bildet dabei jedoch die Grundlage für die Umsetzung des Konzepts. Die Nichteinhaltung der Konvention durch den Zugriff auf private Attribute mithilfe des von Python umbenannten Attributsnamens soll von Entwicklern vermieden werden, um die Integrität der Daten in komplexen Systemen aufrechtzuerhalten. Das hat zur Folge, dass es einen anderen Weg für Entwickler in Python geben muss, mit welchem unter Einhaltung der Richtlinien selbst auf private Attribute zugegriffen werden kann, sollte dies gewünscht sein.

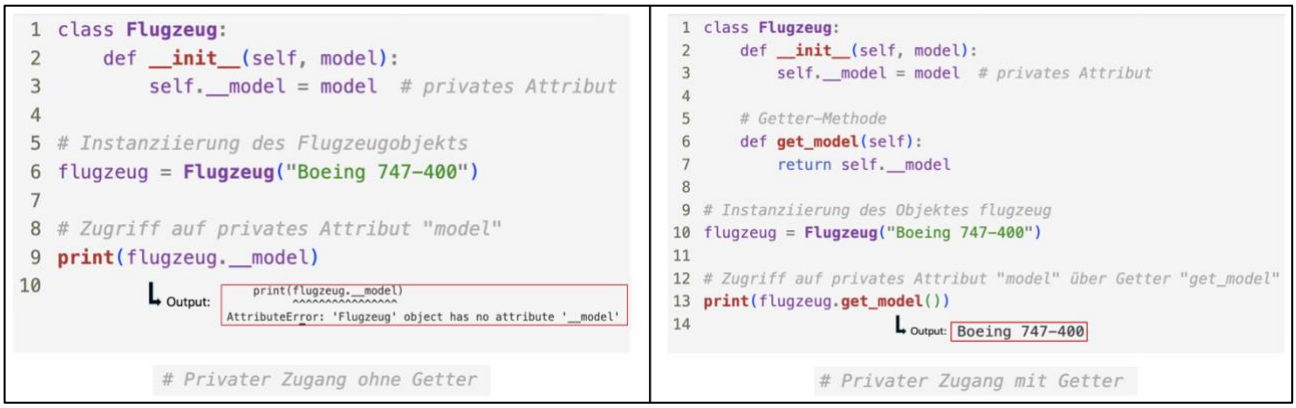
Deshalb gibt es in der objektorientierten Programmierung neben dem Verbergen von Attributen hinter geänderten Namen seitens Python auch die Möglichkeit für Entwickler, den Zugang zu Daten mit sogenannten Gettern und Settern zu kontrollieren und damit Logik zum Schutz von wichtigen Informationen in komplexen Systemen zu implementieren (Steyer, 2018, S. 164).

2.2.5 Getter und Setter

In der objektorientierten Programmierung steht also die Kapselung von Prozessen im Fokus, wodurch Kontrolle über den Zugriff auf Daten in einer Klasse ausgeübt wird und komplexe Systeme in kleine, nachvollziehbare Code-Blöcke mit eigenen Funktionalitäten und Verhaltensweisen aufgeteilt werden. Inwiefern Methoden die Interaktion mit Attributen und Verhaltensweisen von Objekten ermöglichen und abstrahieren, wurde bereits genauer betrachtet. Vor dem Hintergrund, dass in Python, unter Einhaltung der Richtlinien, private Attribute verborgen werden können, lohnt es sich, noch einmal genauer auf eine besondere Art von Methoden zu blicken. Diese bieten die Möglichkeit, Zugang zu privaten Attributen steuerbar für den Entwickler zu machen und werden als Getter und Setter Methoden bezeichnet (Hunt, 2023, S. 269).

Abbildung 9

Gegenüberstellung von Zugriff auf private Attribute mit und ohne Getter



In der Abbildung 9 oberhalb erkennt man auf der linken Seite eine verkürzte Form des bereits betrachteten Beispiels, in welchem der gescheiterte Zugriff auf ein privates Attribut demonstriert wird. Die Klasse Flugzeug wird mit dem privaten Attribut „model“ definiert, es wird ein Objekt namens „flugzeug“ auf Basis dieser Klasse erschaffen und anschließend soll das Attribut „model“ mithilfe der Funktion „print(flugzeug.__model)“ in Zeile 9 in der Ausgabe angezeigt werden. Aufgrund des bereits beschriebenen Name Mangling, also der internen Veränderung von privaten Attributsnamen seitens Python, resultiert das Beispiel ohne eine speziell definiert Getter-Methode in einer Fehlermeldung. Auf der rechten Seite der Abbildung hingegen erkennt man grundsätzlich dasselbe Programm, lediglich um eine Methode namens „get_model(self)“ in Zeile 6 ergänzt, deren einzige Aufgabe darin besteht, den Wert des privaten Attributes „model“ zurückzugeben, sobald sie aufgerufen wird („return self.__model“). In Zeile 13 wird mit „print(flugzeug.get_model())“ das Name Mangling umgangen und daraufhin das Modell angezeigt, da die Methode „get_model“ aufgerufen wird, welche die benötigten Daten findet, liest und anschließend zurückgibt. So kann ein verborgenes, privates Attribut durch einen spezifisch definierten Zugangspunkt, nämlich die Getter-Methode „get_model“, sichtbar gemacht werden. Auf diese Weise kann, im Gegensatz zur Option ohne Getter, ein geschützter und kontrollierter Zugang zu Daten modelliert werden.

Unter Einhaltung der Konvention, dass man nicht mit dem Python-internen Namen auf private Attribute zugreift, bietet die Verwendung einer Getter-Methode den einzigen sicheren Weg, geschützte Informationen zu lesen. In manchen Fällen sollen private Attribute nicht nur gelesen, sondern auch verändert werden. Dies geschieht mit Hilfe von sogenannten Setter-Methoden (Phillips, 2018, S. 357). Setter Methoden ermöglichen neben dem Zugriff und der Veränderung von privaten Attributen eines Objektes ebenfalls die Überprüfung von erlaubten Werten vor der Bearbeitung von Daten, wodurch Werte vor der Änderung gefiltert werden können (Steyer, 2018, S. 163). Wie bereits beim Beispiel zur Getter-Methode (Abbildung 9) demonstriert, kann auf private Attribute nicht durch alleinige Referenz des Objektnamens sowie des Attributs zugegriffen werden, weder um diese zu lesen, noch diese zu ändern. In dem Beispiel in Abbildung 10 unterhalb wird das Programm neben der Getter-Methode „get_model“ (Zeile 7) noch um eine Setter-Methode „set_model“ (Zeile 11) ergänzt. Diese besitzt neben „self“ einen weiteren Parameter, der beim Aufrufen des Setters für die Neuzuweisung des Wertes bereitsteht.

Abbildung 10

Veränderung eines privaten Attributs mit Setter Methode

Innerhalb des Setters „set_model“ wird zu Demonstrationszwecken lediglich der Wert des privaten Attributs „model“ mit dem neuen Wert überschrieben („self.__model = model“, Zeile 12, Abb. 10). Beim Konstruktor „__init__“ in Zeile 3 wird das Attribut „model“ der Klasse Flugzeug mithilfe der beiden Unterstriche als privat definiert, wodurch eine nachträgliche Veränderung nach der Instanziierung eines Objektes nur über einen spezifisch definierten Zugangspunkt erfolgen sollte.

Dieser Zugangspunkt ist der Setter „set_model“, mit welchem das private Attribut in diesem Beispiel mit einem neuen Wert überschrieben wird. Um das Resultat zu verdeutlichen, wird „model“ des Objektes „flugzeug“ mit dem Getter in Zeile 18 gelesen und angezeigt, dann mit dem Setter verändert und anschließend nochmals in Zeile 24 angezeigt. Dabei wird deutlich, dass der Wert des Attributes sich nach dem Aufrufen des Setters in Zeile 21 von „Boeing 747-400“ auf „Airbus A-380“ ändert. Dies zeigt, dass unter Einhaltung der Konventionen mithilfe von Getter- bzw. Setter-Methoden weiterhin mit privaten Attributen interagiert werden kann, wobei der Zugang zu den Informationen nur über spezifisch definierte Zugangsmethoden erfolgt. Des Weiteren können im Besonderen Setter-Methoden dazu genutzt werden, zusätzliche Kontrollinstanzen in das Programm zu implementieren, welche die Veränderung von Objektinformationen nur unter bestimmten Bedingungen zulassen. So kann gefiltert werden, welche Werte als Parameter für Setter-Methoden verwendet werden dürfen, um die Integrität der Daten effektiv zu schützen und die Fehleranfälligkeit des Programms durch versehentliche Änderung von sensiblen Daten zu reduzieren (Steyer, 2018, S. 161).

In der Abbildung 11 unterhalb wird in der Setter-Methode in Zeile 11 eine zusätzliche Kontrollinstanz in Form eines if-statements hinzugefügt. So soll verdeutlicht werden, wie über Getter und Setter sowohl der Zugang, als auch die Veränderung eines Objektattributes in der objektorientierten Programmierung in Python kontrolliert werden kann.

Abbildung 11

Kontrollierte Änderung von Attribut „model“ über Setter „set_model“

```
1  # Zugang zu und Änderung von privaten Objektattributen mit Getter und Setter
2  class Flugzeug:
3      def __init__(self, model):
4          self.__model = model  # privates Attribut
5
6      # Getter-Methode
7      def get_model(self):
8          return self.__model
9
10     # Setter-Methode
11     def set_model(self, model):
12         # Kontrollinstanz mit if-statement: Nur Modelle beginnend mit Airbus bzw. Boeing werden als neues Attribut akzeptiert
13         if model.startswith("Airbus") or model.startswith("Boeing"):
14             self.__model = model
15         # Fehlermeldung bei allen anderen Flugzeugtypen
16         else:
17             raise ValueError("Flugzeugmodell nicht gefunden")
18
19 # Instanziierung des Objektes flugzeug
20 flugzeug = Flugzeug("Airbus A-380")  ⟶ Output: Airbus A-380
21
22 # Zugriff auf privates Attribut "model" von Objekt Flugzeug mithilfe des Getters "get_model()"
23 print(flugzeug.get_model())
24
25 # Änderung des privaten Attributs "model" über Setter-Methode "set_model()" mit zusätzlicher Kontrolle
26 flugzeug.set_model("Antonov AN-225")
27
28 # Zugriff und Anzeige des überschriebenen privaten Attributs "model" mithilfe des Getters "get_model()"
29 print(flugzeug.get_model())
30                                  ↳ Output: ValueError: Flugzeugmodell nicht gefunden
```

In der Klasse Flugzeug gibt es in diesem Beispiel nur das Attribut „model“, welches innerhalb des Konstruktors __init__ in Zeile 3 definiert wird. Dieses Attribut ist privat und kann nun in jedem auf Basis der Klasse erschaffenen Objekt enthalten sein. Danach wird der Getter mit „get_model“ definiert, der das Lesen und die Anzeige des privaten Attributes ermöglicht, sobald ein Objekt erschaffen wurde und dieses enthält. In Zeile 11 bis Zeile 17 wird nun der Setter definiert, über welchen das private Attribut „model“ in einem Objekt nicht nur gelesen, sondern auch verändert werden kann. Dafür muss es gewisse Voraussetzungen erfüllen, welche in Form eines if-statements beschrieben werden (Zeile 13, Abbildung 11). Konkret muss die Voraussetzung erfüllt sein, dass das geänderte Flugzeugmodell mit dem Namen „Boeing“ oder „Airbus“ beginnen muss, dafür sorgt die Funktion „startswith()“ (Zeile 13, Abbildung 11). Diese in Python enthaltene Funktion überprüft Anfänge von Strings, also Daten in Textform, und vergleicht sie anschließend mit den genannten Parametern. Ist die Voraussetzung erfüllt und der String-Anfang wird von der Funktion „startswith()“ erkannt, wird das Attribut „model“ überschrieben und beinhaltet nun den neuen Wert.

Ist die Voraussetzung nicht erfüllt, wird dem Nutzer eine Fehlermeldung angezeigt, welche in Zeile 17 mit der Funktion „raise ValueError()“ implementiert wurde und als Parameter einen einfachen Text enthält („Flugzeugmodell nicht gefunden“). Wie in der Abbildung 11 demonstriert, wird versucht, das Flugzeugmodell von „Airbus A-380“ zu „Antonov AN-225“ (Zeile 26, Abbildung 11) zu ändern, was die Voraussetzungen im if-statement des Setters „set_model()“ nicht erfüllt. Der Output resultiert deshalb schlussendlich in der vordefinierten Fehlermeldung. Eine solche Überprüfung der vom Nutzer eingegebenen Daten könnte dabei bereits im Konstruktor des Programms erfolgen, wodurch eine Kontrollinstanz bereits bei der Erschaffung eines Objektes auf Basis der Klasse „Flugzeug“ implementiert wird. In diesem Beispiel erfolgt dieser Schritt zu Demonstrationszwecken erst nach der Instanziierung des ersten Objektes „flugzeug“, bei der Veränderung des Attributes. Wichtig zu erwähnen ist hier, dass nicht nur Attribute, sondern auch Methoden, mithilfe derselben Vorgehensweise als privat gekennzeichnet werden können und anschließend durch Name Mangling von Python verborgen werden. So kann in der Praxis beispielweise eine Methode als Zugangspunkt für alle verborgenen Methoden genutzt werden, sollte dies erwünscht sein. Mithilfe von speziellen Methoden, die einmalig in der Klasse definiert werden, kann also Kontrolle über all die daraus resultierenden Objekte und ihre Werte ausgeübt werden.

Zusammenfassend lässt sich sagen, dass das Konzept der Kapselung in der objektorientierten Programmierung in Python mehrere Ziele verfolgt. Allen voran steht der Schutz der Daten, da interne Informationen eines Objektes von außen nur mithilfe von speziell definierten Methoden überschrieben werden können, was die unerwünschte bzw. versehentliche Veränderung von Daten vermeiden soll. Kapselung hängt aber auch mit dem Konzept der Abstraktion in der objektorientierten Programmierung zusammen. Anhand der Beispiele wird deutlich, dass ein Nutzer mithilfe von Setter-Methoden von außen auf private Attribute eines Objektes zugreifen und diese auch verändern kann, ohne dabei genau wissen zu müssen, welche Schutzmechanismen innerhalb des Setters definiert werden, um die Integrität der Daten zu schützen. Python kann mithilfe der klasseninternen Definition von Attributen und Methoden nach der Erschaffung eines Objektes stets erkennen, um welche Informationen innerhalb des Programmes es sich gerade handelt. So kann durch Referenz des Objektnamens sowie des Attributnamens oder des Methodennamens direkt auf Daten zugegriffen werden, wie zum Beispiel bei „flugzeug.model“ (Abbildung 8) oder „flugzeug.get_model()“ (Abbildung 9), um diese selbst bei komplexen Programmen mit vielen Objekten intuitiv und in abstrahierter Form verändern zu können. Außerdem soll Kapselung bewirken, dass Programme aufgrund ihres modularen Aufbaus leichter gewartet und getestet werden können, da Zuständigkeitsbereiche in Klassen samt ihrer Objekte festgehalten werden und deshalb in einer nachvollziehbaren und lesbaren Struktur aufgebaut sind. Soll das Programm um neue Funktionalitäten erweitert werden, kann der modulare Aufbau aber auch zu Herausforderungen führen. Bleibt man dem Prinzip der Kapselung treu, kann dies dazu führen, dass neue Klassen vielleicht in einigen Punkten mit bereits definierten Klassen übereinstimmen, dabei aber auch einige Zusatzfunktionen beinhalten.

Eines der Ziele des modularen Aufbaus eines Programms besteht in der übersichtlichen und direkten Erweiterung mithilfe von Bausteinen, die eigene Zuständigkeitsbereiche in einem System widerspiegeln, ohne dabei Code duplizieren zu müssen. Gleichzeitig ist es aber nicht zielführend, Objekte und ihre Eigenschaften mithilfe einer einzigen Klasse darzustellen und darin alle Funktionalitäten festzuhalten. Objekte können in der Realität oft identische Merkmale besitzen, sich aber trotzdem in einigen Punkten grundlegend voneinander unterscheiden. Anstatt eine Klasse mit unzähligen, verschiedenen Attributen zu definieren, welche anschließend bei der Instanziierung mit einem Wert versehen werden, gibt es in der objektorientierten Programmierung die sogenannte Vererbung (Phillips, 2018, S. 20). Ein Modell, mit welchem einzelne Klassen mit Abhängigkeiten voneinander kreiert werden können, ohne dabei erneut alle Eigenschaften und Zustände definieren zu müssen. Nur auf eine solche Weise können realitätsnahe, komplexe Programme geschrieben werden, in welchen voneinander abhängige Klassen unter Berücksichtigung ihrer unterschiedlichen Funktionen modelliert werden können. Auf welche Weise die Vererbung genau das erreicht und warum sie eines der wichtigsten Konzepte der objektorientierten Programmierung darstellt, wird im Folgenden genauer betrachtet.

2.2.6 Vererbung

Ein komplexes Programm zu erschaffen, in welchem zahlreiche Prozesse in eigene, kleinere Aufgabenbereiche aufgeteilt werden, geht mit der Herausforderung einher, die einzelnen Bausteine zu einem funktionierenden Gesamtkonstrukt zu bündeln. Genauso wie man den Begriff Vererbung aus der Realität kennt, beispielsweise bei Eltern und Kindern, beschreibt sie in der objektorientieren Programmierung das Erben von Daten einer Klasse von einer Anderen (Hunt, 2023, S. 237). Die Klasse, welche die Daten an eine Weitere vererbt, wird auch als Elternklasse bezeichnet, während die erbende Klasse als Kindklasse bezeichnet wird (Phillips, 2018, S. 20). Eigenschaften und Verhaltensweisen, die in einer Klasse definiert wurden, können in der Definition einer neuen Klasse wiederverwendet und um neue Daten ergänzt werden, um Attribute und Methoden wiederverwendbar zu machen und Duplikationen zu vermeiden (Hunt, 2023, S. 249). Es erleichtert außerdem die Wartung von Programmen, da bei möglichen Problemen nicht die Definitionen in jeder Klasse kontrolliert werden müssen, sondern der Blick zunächst auf die Elternklassen gerichtet werden kann, auf deren Basis alle Kinderklassen erschaffen wurden. In der Abbildung 12 unterhalb wird die Vererbung anhand eines Beispiels demonstriert:

Abbildung 12

Aufbau eines Programms mit einer Elternklasse und zwei Kinderklassen

```
1  class Flugzeug:
2      def __init__(self, model, flight_number, status):
3          self.model = model
4          self.flight_number = flight_number
5          self.status = status
6
7  class Passagierflugzeug(Flugzeug):
8      def __init__(self, model, flight_number, status, passagiere):
9          super().__init__(model, flight_number, status)
10         self.passagiere = passagiere
11
12 class Transportflugzeug(Flugzeug):
13     def __init__(self, model, flight_number, status, frachtgewicht):
14         super().__init__(model, flight_number, status)
15         self.frachtgewicht = frachtgewicht
```

In der Abbildung 12 kann man zunächst die bereits bekannte Klasse Flugzeug mit ihren drei Attributen erkennen (model, flight_number, status), die in Zeile 1 bis 5 des Programms definiert werden. Die Klasse Flugzeug stellt im Kontext der Vererbung die Elternklasse dar, auch genannt Superklasse oder Basisklasse, da sie der Ausgangspunkt für alle weiteren Klassen ist und im Folgenden ihre Attribute an alle weiteren Klassen vererben wird (Hunt, 2023, S. 238).

Die beiden Klassen „Passagierflugzeug“ (Zeile 7) sowie „Transportflugzeug“ (Zeile 12) sind im Rahmen der Vererbung nun besonders interessant und werden betrachtet, um ein Verständnis dafür zu bekommen, in welcher Beziehung sie mit der Klasse „Flugzeug“ stehen und warum das Konzept der Vererbung ein so wichtiges Instrument der objektorientierten Programmierung darstellt. In der Abbildung 13 ist die Klasse „Passagierflugzeug“ hervorgehoben, die im ersten Moment ähnlich aufgebaut ist wie alle bisher betrachteten Klassen, jedoch mit einigen grundlegenden Unterschieden.

Abbildung 13

Kindklasse Passagierflugzeug

```
 7 class Passagierflugzeug(Flugzeug):
 8     def __init__(self, model, flight_number, status, passagiere):
 9         super().__init__(model, flight_number, status)
10         self.passagiere = passagiere
```

In Zeile 7 wird nun als Parameter hinter dem Klassennamen „Passagierflugzeug“ der Name der Elternklasse „Flugzeug“ genannt, wodurch Python erkennt, dass es sich hierbei um eine Kindklasse handelt, die auf Basis der in den Klammern genannten Elternklasse gebildet wird („class Passagierflugzeug(Flugzeug)“). In Zeile 8 der Abbildung 13 wird anschließend der Konstruktor „__init__“ definiert, welcher dieselben Attribute wie die Elternklasse enthält, also „model“, „flight_number“ sowie „status“. Es wird aber gleichzeitig noch ein weiteres Attribut hinzugefügt, dass so bei der Elternklasse nicht vorkommt, nämlich „passagiere“. Dieser Konstruktor hat die Aufgabe, bei jedem Objekt, dass auf Basis der Klasse Passagierflugzeug erschaffen wird, die Attribute „model“, „flight_number“, „status“ und „passagiere“ an das Objekt weiterzugeben, sodass diese anschließend Daten zu genau diesen Attributen aufnehmen können. Im Gegensatz zu den bisher betrachteten Klassen fehlen hier die Definitionen der einzelnen Attribute innerhalb des Konstruktors mithilfe der Syntax: „self.model = model“, „self.flight_number = flight_number“ sowie „self.status = status“. In der Abbildung 13 wird lediglich das neue Attribut „passagiere“ in Zeile 10 formuliert, erkennbar anhand der Syntax „self.passagiere = passagiere“. Da die anderen Attribute bereits in der Elternklasse „Flugzeug“ definiert wurden, müssen sie nur in den Klammern nach dem Konstruktor „__init__“ (Zeile 8, Abbildung 13) genannt, anschließend in der Klasse aber nicht nochmal definiert werden. Ermöglicht wird dies durch den Vorgang in Zeile 9, da mithilfe der Formulierung „super().__init__(model, flight_number, status)“ ein Bezug zur Elternklasse und somit Superklasse „Flugzeug“ hergestellt wird, durch welchen Python erkennt, dass die Attribute dort bereits definiert wurden und nun weiter an die eben erstellte Kindklasse vererbt werden. Dies führt dazu, dass die Kindklasse „Passagierflugzeug“ alle Attribute von der Elternklasse erbt, aber auch eigene, spezifische Daten festhalten kann, wie beispielsweise ein Attribut „passagiere“, welches nur bei Passagierflugzeugen relevant ist.

Als Nächstes wird das Programm um eine weitere Kindklasse namens „Transportflugzeug“ ergänzt: Abbildung 14 veranschaulicht die Kindklasse „Transportflugzeug“, die genauso wie „Passagierflugzeug“ auf Basis der Elternklasse „Flugzeug“ definiert wird.

Abbildung 14

Erschaffung der Kindklasse „Transportflugzeug“

```
12 class Transportflugzeug(Flugzeug):
13     def __init__(self, model, flight_number, status, frachtgewicht):
14         super().__init__(model, flight_number, status)
15         self.frachtgewicht = frachtgewicht
```

Anstatt die Anzahl der Passagiere mithilfe des Attributs „passagiere“ festzuhalten, wird hier neben Informationen zu „model“, „flight_number“ und „status“ die Eigenschaft „frachtgewicht“ als neuer Parameter des Konstruktors __init__ in Zeile 13 genannt. Während bei einem Passagierflugzeug die Passagieranzahl wichtig ist, steht bei Transportflugzeugen das Gewicht der Transportgüter im Vordergrund. Ansonsten unterscheiden sich die Kindklassen nicht voneinander. In beiden Fällen werden einerseits Attribute wie das Flugzeugmodell, Flugnummer oder auch der aktuelle Status von der Elternklasse „Flugzeug“ weitergegeben, andererseits eigene, spezifische Attribute für den jeweiligen Flugzeugtyp hinzugefügt (Zeile 15, Abbildung 14: „self.frachtgewicht = frachtgewicht“). Erschafft man anschließend drei verschiedene Objekte auf Basis der Elternklasse sowie der beiden Kindklassen, wird deutlich, in welcher Weise die Vererbung dafür genutzt werden kann, Zusammenhänge zwischen Klassen strukturiert zu modellieren und weitere Funktionalitäten in ein Programm zu integrieren. Gleichzeitig soll demonstriert werden, dass Vererbung nicht nur für die Weitergabe von Informationen in Form von Attributen genutzt wird, sondern ebenfalls die Weitervererbung von Methoden ermöglicht.

Abbildung 15

Elternklasse „Flugzeug“

Elternklasse/Superklasse/Basisklasse

```
class Flugzeug:
    def __init__(self, model, flight_number, status):
        self.model = model
        self.flight_number = flight_number
        self.status = status

    def __str__(self):
        return f"Flugzeugmodell: {self.model}, Flugnummer: {self.flight_number}, Status des Flugzeugs: {self.status}"
```

Unterklasse/Subklasse/Kindklasse

```
class Passagierflugzeug(Flugzeug):
    def __init__(self, model, flight_number, status, passagiere):
        super().__init__(model, flight_number, status)
        self.passagiere = passagiere

    def __str__(self):
        return f"{super().__str__()}, Passagiere: {self.passagiere}"
```

Unterklasse/Subklasse/Kindklasse

```
class Transportflugzeug(Flugzeug):
    def __init__(self, model, flight_number, status, frachtgewicht):
        super().__init__(model, flight_number, status)
        self.frachtgewicht = frachtgewicht

    def __str__(self):
        return f"{super().__str__()}, Frachtgewicht: {self.frachtgewicht}"
```

In Abbildung 15 wird der grundsätzliche Aufbau der Vererbung anhand der Klassen aus dem Flughafenbeispiel veranschaulicht. Wie man erkennen kann, bildet die Elternklasse die Basis der gesamten Vererbung, da in ihr die Eigenschaften und Vorgänge definiert werden, welche dann an alle daraus entstehenden Kindklassen weitervererbt werden. Jede mögliche Kindklasse wird die Attribute „model“, „flight_number“ sowie „status“ erhalten, da jedes Flugzeug, egal ob Passagier- oder Transportflugzeug, über ein eigenes Modell oder einen für sich spezifischen Status verfügt. Anstatt die Attribute in den Kindklassen einzeln neu definieren zu müssen, werden sie über einen Verweis auf die Superklasse mithilfe der Syntax „super().__init__(model, flight_number, status)“ in die Kindklassen integriert. Die einzigen Attribute, die explizit in der Kindklasse definiert werden, sind die klassenspezifischen Eigenschaften wie beispielsweise „passagiere“ in „Passagierflugzeug“. Darüber hinaus wird die Methode „__str__“ wiederholt dafür verwendet, die Informationen eines Objektes nach erfolgreicher Instanziierung für den Nutzer in Textform darzustellen.

Die Methode „__str__“ demonstriert hier darüber hinaus ein weiteres Konzept der Vererbung: Methoden können in der objektorientierten Programmierung ebenso vererbt werden wie Attribute und anschließend von Kindklassen für unterschiedliche Zwecke implementiert werden. In der Elternklasse „Flugzeug“ wird die „__str__“-Methode definiert, mithilfe der Syntax „f“Flugzeugmodell: {self.model}, Flugnummer: {self.flight_number}, Status des Flugzeugs: {self.status}““. Auf diese Weise werden schrittweise all die Attribute eines Objektes gelesen und dann in das vorgegebene Format gebracht. Blickt man nun in die beiden Kindklassen „Passagierflugzeug“ sowie „Transportflugzeug“, findet man unterhalb des Konstruktors mit den vererbten Attributen sowie dem klassenspezifischen neuen Attribut („passagiere“ & „frachtgewicht“) bei Beiden die Methode „__str__“.

Die Struktur der „__str__“-Methode unterscheidet sich dabei aber von dem, was in der Elternklasse zu sehen ist. Anstatt die von der Elternklasse „Flugzeug“ geerbten Attribute noch einmal zu nennen, beinhaltet der F-String eine Referenz zur „__str__“-Methode der Elternklasse. Als F-String wird das angesprochene Format bezeichnet, mit welchem Werte und Funktionen dynamisch in Textform dargestellt werden können. Indem man Namen von Variablen, Attributen, Funktionen oder auch Methoden in geschweifte Klammern setzt, können verschiedene Datentypen in einen Text eingesetzt werden, um anschließend ein einheitliches Format mit allen relevanten Informationen und Daten zu erhalten. Dies wird im Anschluss als Resultat der „__str__“-Methode an Python zurückgegeben und kann nach erfolgreicher Instanziierung eines Objektes, welches die „__str__“-Methode enthält, für den Nutzer mithilfe einer print()-Funktion in Textform dargestellt werden. Wie man anhand der Abbildung 16 erkennen kann, enthält die erste geschweifte Klammer bei beiden Kindklassen eine Referenz zur „__str__“-Methode in der Elternklasse in Form von „{super().__str__()}“.

Abbildung 16

Die __str__ Methode innerhalb der beiden Kindklassen „Passagierflugzeug“ und „Transportflugzeug“

```
def __str__(self):
    return f"{super().__str__()}, Passagiere: {self.passagiere}"
```

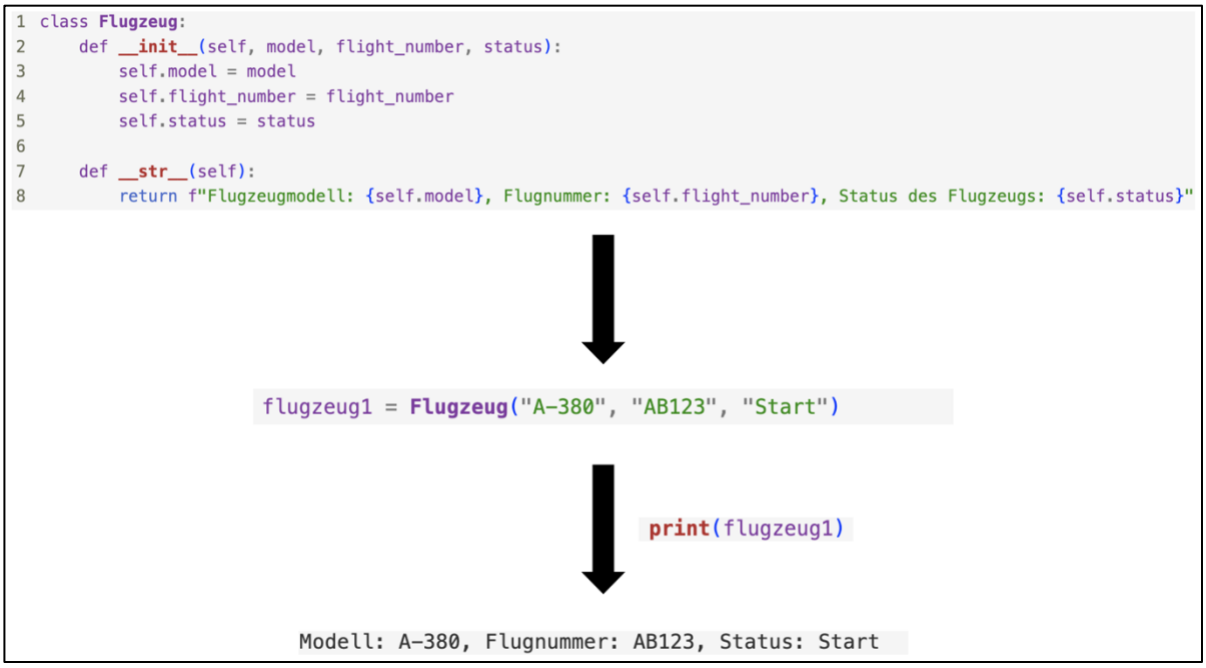
```
def __str__(self):
    return f"{super().__str__()}, Frachtgewicht: {self.frachtgewicht}"
```

Genauso wie bei der Vererbung von Attributen wird Python auf diese Weise mitgeteilt, dass die „__str__“-Methode bereits in der Elternklasse definiert wurde und dabei spezifische Informationen enthält, welche nun auch an die Kindklasse weitergegeben werden sollen. Obwohl die eben beschriebenen spezifischen Informationen in diesem Beispiel aus der Ausgabe von Attributen in einer einheitlichen Textform bestehen, könnten auch weitaus komplexere Methoden, beispielsweise Getter und Setter mit eigenen Kontrollinstanzen, von Klasse zu Klasse weitergegeben werden. Da die Attribute der Elternklasse bereits mithilfe des Konstruktors, welcher die Anweisung „super().__init__(model, flight_number, status)“ enthält, an die Kindklasse weitergegeben wurden, erkennt Python, um welche Daten es sich handelt. Der Parameter „self“, dessen Rolle bereits beschrieben wurde, stellt für Python einen lesbaren Bezug zum Objekt her. In den „__str__“ Methoden der Kindklassen wird nach der Referenz auf die Elternklasse in der ersten geschweiften Klammer jeweils noch das klassenspezifische Attribut in den F-String aufgenommen, welches erst mit der Instanziierung eines Objektes auf Basis der Kindklassen erschaffen wird. Im Falle der Kindklasse „Passagierflugzeug“ ist es das Attribut „passagiere“, welches in Form von „{self.passagiere}“ in den F-String aufgenommen wird, bei der Kindklasse „Transportflugzeug“ ist es das Attribut „frachtgewicht“ mithilfe von {self.frachtgewicht}“.

Erschafft man abschließend jeweils ein Objekt auf Basis von „Flugzeug“, „Transportflugzeug“ und „Passagierflugzeug“, bekommt man einen Überblick, welchen Einfluss die Vererbung auf Klassen, Objekte und die darin enthaltenen Informationen ausüben kann und weshalb sie eine der grundlegenden Konzepte in der objektorientierten Programmierung darstellt.

Abbildung 17

Instanziierung des Objektes „flugzeug1“ auf Basis der Elternklasse „Flugzeug“

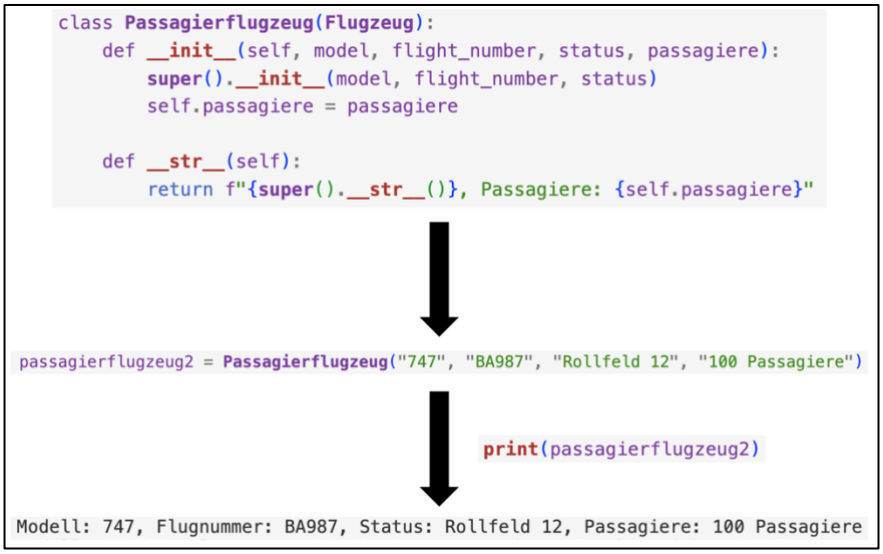


Die Elternklasse „Flugzeug“ dient in der Abbildung 17 als Blaupause zur Erschaffung eines Objektes. Mithilfe der Syntax „Flugzeug(„A-380“, „AB123“, „Start“) wird ein Objekt instanziiert, wobei die Parameter in Klammern die benötigten Informationen für die in der Klasse definierten Attribute „model“, „flight_number“ und „status“ darstellen. Anschließend wird das instanziierte Objekt in einer Variable namens „flugzeug1“ gespeichert und dann mithilfe der Funktion „print(flugzeug1)“ in der Ausgabe angezeigt. Das Resultat beinhaltet die bei der Instanziierung genannten Attributsdaten sowie die Darstellung dieser Informationen in Form der in der Klasse definierten „__str__“-Methode. Dieser Vorgang sollte inzwischen klar sein, viel interessanter ist hier die Erschaffung eines weiteren Objektes auf Basis der ersten Kindklasse „Passagierflugzeug“.

In der Abbildung 18 werden mithilfe der Vererbung Attribute sowie die Methode „__str__“ von der Elternklasse geerbt.

Abbildung 18

Instanziierung des Objektes „passagierflugzeug2“ auf Basis der Kindklasse „Passagierflugzeug“

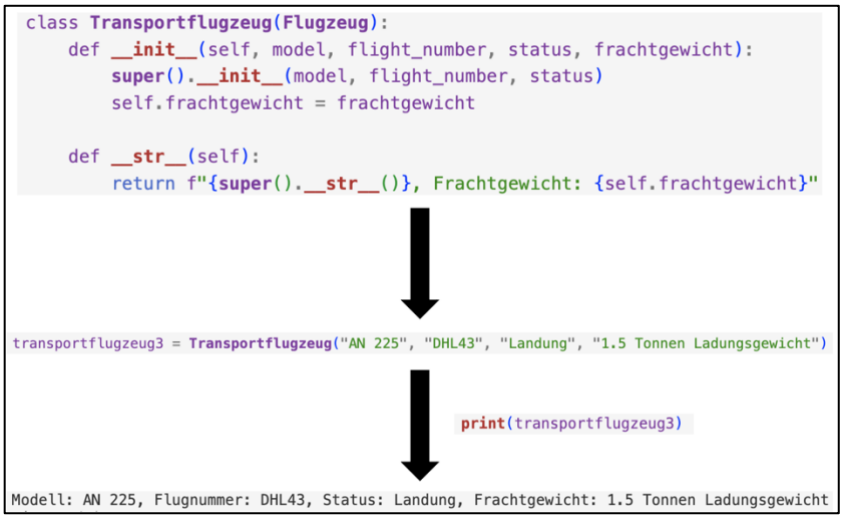


Durch „super().__init__(model, flight_number, status)“ erhält die Kindklasse alle wichtigen Attribute von der Elternklasse „Flugzeug“, weshalb diese hier nicht mehr einzeln mit der Syntax „self.attributsname = attributsname“ definiert werden müssen. Lediglich die für die Kindklasse spezifische Eigenschaft „Passagiere“ wird hier zusätzlich in Form von „self.passagiere = passagiere“ definiert. Somit muss bei der Erschaffung eines Objektes darauf geachtet werden, dass Objekte der Kindklasse insgesamt vier Parameter aufnehmen. Dies demonstriert der nächste Schritt, da bei der Instanziierung in Form von „Passagierflugzeug(„747“, „BA987“, „Rollfeld 12“, „100 Passagiere“) ein zusätzlicher Parameter im Vergleich zur Elternklasse genannt werden muss. Fehlt der vierte Parameter hier, erhält der Nutzer eine Fehlermeldung. Anstatt also in der Kindklasse wieder alle Attribute einzeln definieren zu müssen, kann dieser Schritt mithilfe der Vererbung verkürzt werden, wodurch die Kindklasse in einem übersichtlichen Stil um ergänzende Informationen erweitert werden kann. Auch die in der Elternklasse definierte „__str__“-Methode wird in ihrer Vollständigkeit an die Kindklasse weitergegeben. Wie man anhand des Beispiels erkennen kann, reicht die Referenz in der „__str__“-Methode der Kindklasse in Form von „return f“{super().__str__()}“ aus, um sowohl auf die Formatierung des f-Strings, als auch auf die darin enthaltenen Verweise auf bestimmte Attribute zuzugreifen. Daraufhin wird der f-String um ein weiteres Attribut ergänzt. Dies geschieht in Form von „Passagiere: {self.passagiere}“, direkt nach dem Verweis auf die „__str__“-Methode in der Elternklasse. Anschließend wird das Objekt in der Variable „passagierflugzeug2“ gespeichert und mithilfe von print(passagierflugzeug2) angezeigt.

Das Resultat verdeutlicht, dass sowohl die Attribute als auch die „__str__“-Methode in ihrer Gesamtheit an die Kindklasse weitergegeben wurden, wo sie auf abstrahierte Weise um klassenspezifische Informationen ergänzt werden können. Dies ermöglicht die Erschaffung von Programmen, in welchen einzelne Prozesse in Code-Blöcken zwar voneinander abgekapselt werden, nichtsdestotrotz in Wechselwirkung zueinander stehen. Mithilfe der Vererbung können neue Eigenschaften und Funktionalitäten effektiv implementiert werden, ohne dabei die Elternklassen verändern zu müssen, da zusätzliche Informationen innerhalb der neuen Klassen definiert werden können, während auf bereits bekannte Daten durch Referenz auf vorhandene Klassen zugegriffen werden kann. Im Umkehrschluss kann gegen eventuelle Fehler im Programm vorgegangen werden, indem man zunächst die in der Elternklasse implementierten Methoden und Attribute untersucht, da von dort ausgehend viele der grundlegenden Prozesse in den Kindklassen formuliert werden. Die Implementierung neuer Objekte anhand der Elternklasse „Flugzeug“ kann beliebig oft wiederholt werden.

Abbildung 19

Instanziierung des Objektes „transportflugzeug3“ auf Basis der Kindklasse „Transportflugzeug“



In Abbildung 19 wird eine neue Kindklasse „Transportflugzeug“ definiert, die das eigene Attribut namens „frachtgewicht“ enthält, ansonsten aber genauso aufgebaut ist wie die vorher beschriebene Kindklasse „Passagierflugzeug“. Da es für ein Transportflugzeug vereinfacht dargestellt nicht von Relevanz ist, wie viele Passagiere an Bord sind, sondern viel mehr das Gewicht an Gütern in Tonnen interessant ist, wird das Attribut „frachtgewicht“ als klasseninterne Eigenschaft definiert. Auch hier sind Informationen zum Flugzeugmodell etc. ebenfalls wichtig für die objektorientierte Darstellung in einem Programm, weshalb diese von der Elternklasse vererbt und hier genutzt werden. Deshalb enthält das Objekt bei der Instanziierung vier Parameter, die wiederum bei der Erschaffung eines Objektes mit neuen Daten versehen werden können.

Die „__str__“-Methode demonstriert die erfolgreiche Erschaffung eines Transportflugzeug-Objektes in einer übersichtlichen Form und verdeutlicht, dass durch einmaliges Definieren einer Methode immer wieder darauf zugegriffen werden kann. Der Basiszustand der Methode wird um klassenspezifische Veränderungen ergänzt (hier: return f“{super().__str__()}, **Frachtgewicht: {self.frachtgewicht}“).** Beide Kindklassen beinhalten also ihre eigene Variante der Methode und überschreiben dabei die inneren Details, welche in der Elternklasse definiert wurden, mit neuen Daten. Daraufhin ergänzen sie diese um die klassenspezifischen Attribute. Diese Art der Überschreibung von Methoden spielt eine zentrale Rolle in der objektorientierten Programmierung. Der wiederholte Aufruf einer Methode in verschiedenen Kindklassen mit unterschiedlichen Daten wird in der objektorientierten Programmierung als Polymorphismus bezeichnet und bildet zusammen mit den Konzepten der Abstraktion, Kapselung und Vererbung die vier Grundpfeiler dieses Programmierparadigmas (Steyer, 2018, S. 170). Vererbte Methoden können außerdem nicht nur zusätzliche Kontrollpunkte enthalten und diese weitergeben, sondern auch verschiedene Sichtbarkeitsstufen. Werden bestimmte Eigenschaften in einer Klasse als privat gekennzeichnet, können Getter- und Setter-Methoden daraufhin auch an Kindklassen weitervererbt werden, wodurch der Zugang über alle Kindklassen hinweg in einer konsistenten Weise formuliert werden kann. Abschließend sollte erwähnt werden, dass von mehreren Elternklassen gleichzeitig geerbt werden kann. Die hier beschriebene Struktur von einer Elternklasse auf zwei Kindklassen kann tatsächlich um Einiges komplexer ausfallen, falls tiefere Zusammenhänge zwischen Klassen in einem Programm beschrieben werden. Der genauere Ablauf einer solchen Vorgehensweise wird im Rahmen dieser Arbeit jedoch nicht mehr diskutiert.

3. Fazit

Die objektorientierte Programmierung ist in Python eines der wichtigsten Paradigmen und enthält viele verschiedene Mechanismen, mit denen Programme in einer spezifischen Struktur dargestellt werden können. Durch einige grundlegende Konzepte wie der Definition von Klassen und damit zusammenhängend Objekten kann auf abstrahierte, realitätsnahe Weise in einem Programm dargestellt werden, wie Prozesse in der realen Welt ablaufen und welche Informationen dabei relevant sind. Methoden dienen dabei als Zugangspunkt für Informationen und können gleichzeitig auf dynamische Weise mit Daten interagieren und diese, falls nötig, verändern. Gleichzeitig erhöhen sie die Integrität der Daten, da sie genutzt werden können, um kontrollierte Zugriffe auf sensible Daten durchzuführen. Sie bieten außerdem die Möglichkeit, Kontrolle über die Definition und Veränderung von Attributen in Objekten auszuüben, indem sie zur Beschreibung von Voraussetzungen genutzt werden. Das Maß an Kontrolle über sich verändernde Daten wird in Python verstärkt durch die Möglichkeit der Bestimmung von Sichtbarkeitsstufen bei Informationen (u.A. public und __private). Durch die Vererbung können all die in einer Klasse festgehaltenen Prozesse und Informationen direkt von einer Klasse an eine Weitere vererbt werden, wodurch die Implementierung neuer Klassen und Objekte erleichtert wird und in einer einheitlichen, übersichtlichen Form dargestellt werden kann. Das im Rahmen dieser Arbeit besprochene Beispiel des Flughafens mit diversen Flugzeugobjekten ist zu Demonstrationszwecken in einer sehr simplen Form dargestellt. Nichtsdestotrotz bietet die objektorientierte Programmierung eine vielfältig einsetzbare Grundlage für die Organisation und Strukturierung von Daten in einem Programm und kann deshalb beispielsweise in der Verwaltung von Passagierinformationen seitens der Fluggesellschaften in realen Anwendungen genutzt werden. Nachdem im Laufe der Arbeit einige der Stärken der objektorientierten Programmierung in Python erläutert wurden, gehen einige der Möglichkeiten auch mit potentiellen Herausforderungen einher, auf welche abschließend eingegangen werden soll. Einer der Nachteile ist die bereits angesprochene Datenintegrität bei privaten sowie öffentlichen Informationen in Python, da diese nur durch Konventionen umgesetzt wird. Sollte sich ein Softwareentwickler dafür entscheiden, die Richtlinien zu ignorieren, kann er bestimmte Vorgänge in Python umgehen und auf private Attribute zugreifen. Die Einhaltung der Konventionen liegt somit bei den Entwicklern und kann nicht durch Schutzmechanismen erzwungen werden. Die Vererbung bietet auf der einen Seite ein Instrument zur Darstellung von Beziehungen zwischen Klassen, kann aber bei komplexen Systemen mit mehreren Elternklassen und deren Kindklassen unübersichtlich und schwer nachvollziehbar werden (Monday et al., 2021, S. 28). Ein Beispiel für die Herausforderung der multiplen Vererbung beschreibt das sogenannte Diamanten-Problem, welches auftritt, sobald vier Klassen in Wechselwirkung zueinander stehen (Monday et al., 2021, S. 29).

Dabei kann es dazu kommen, dass zwei Klassen (B & C) eine Methode von einer Elternklasse A erben und diese dann wiederum an eine vierte Klasse D weitervererben, in welcher die Frage geklärt werden muss, ob die vererbte Methode aus Klasse B oder Klasse C priorisiert wird (Monday et al., 2021, S. 29). Obwohl es Lösungen für dieses Problem gibt, wird durch das Diamanten-Problem deutlich, dass in Python aufgrund der vielen möglichen Herangehensweisen für Entwickler ein besonderes Maß an Struktur erforderlich ist. Nur auf diese Weise können komplexe Programme übersichtlich und mit einer tiefgehenden Vererbungshierarchie leistungsstark gestaltet werden. Das objektorientierte Programmierparadigma bietet Rahmenbedingungen, die den Entwickler dabei unterstützen sollen, eben dieses besondere Maß an Struktur zu erreichen. Ein weiteres Problem, dass durch den Einsatz von zahlreichen Klassen und Objekten entstehen kann, ist ein hoher Leistungs- und Speicheraufwand für den Computer, welcher bei rechenintensiven Prozessen mit vielen Informationen sogar noch weiter verstärkt wird. Nichtsdestotrotz bietet die objektorientierte Programmierung ein konzeptionelles Fundament, welches bei der richtigen Umsetzung die Grundlage für die Erschaffung von wartbarer, erweiterbarer und strukturierter Software bietet. Unter Einhaltung der Konventionen bietet dieses besondere Programmierparadigma aufgrund der vorgestellten Abstraktion, Kapselung, Vererbung und der demonstrierten Anwendung des Polymorphismus ein vielfältig einsetzbares Instrument, mit welchem Zusammenhänge und Daten realitätsnah dargestellt werden können, was es zu einer sehr beliebten Vorgehensweise für Entwickler bei der Erschaffung von komplexer Software in Python macht. Voraussetzung für eine erfolgreiche Umsetzung ist die Einhaltung der Konventionen und der richtige Umgang mit den besprochenen möglichen Herausforderungen, welche dieses Programmierparadigma mit sich bringt.

Quellenverzeichnis

Klein, Bernd . *Einführung in Python 3: Für Ein- und Umsteiger*, Deutschland, Carl Hanser Verlag GmbH Co KG., 2021

Steyer, Ralph: *Programmierung in Python: Ein kompakter Einstieg für die Praxis*, Wiesbaden, Deutschland: Springer-Verlag, 2018

PYPI · The Python Package Index, (Software Verzeichnis), 2024, PyPI: https://pypi.org/, [abgerufen am 28.08.2024]

Prof. Dr. Uwe Kastens : *ProgrammierParadigma*, (Verzeichnis), 08.04.2019, https://wi-lex.de/index.php/lexikon/technologische-und-methodische-grundlagen/sprache/programmiersprache/programmierparadigma/ [abgerufen am 02.09.2024]

Eze, Monday, Okunbor, Charles & Chukwudum, Umoke: *Studies in object-oriented programming backbone implementations*, 30.09.2021, Zenodo (CERN European Organization For Nuclear Research), https://zenodo.org/records/5555908, Seite 28-30 [abgerufen am 12.09.2024]

Brborich, Wladymir, Oscullo, Bryan, Lascano, Jorge Edison & Clyde, Stephen*: An Observational Study on the Maintainability Characteristics of the Procedural and Object-Oriented Programming Paradigms*, (Wissenschaftliches Paper), 10.11.2020, (Nr. 2377-570X). IEEE Conference, https://doi.org/10.1109/CSEET49119.2020.9206213 [abgerufen am 18.09.2024]

Neumann, Markus: *C Programmieren für Einsteiger: Der leichte Weg zum C-Experten*, Deutschland, BMU Verlag, 29.01.2020

Phillips, Dusty: *Python 3 Object oriented Programming*, Birmingham, UK, Mumbai Packt, Oktober 2018

Lee, Graham: *Modern Programming: Object Oriented Programming and Best Practices: Deconstruct object-oriented programming and use it with other programming paradigms to build applications*, Birmingham, UK Packt Publishing Ltd., 2019

Gabbrielli, Maurizio, Martini, Simone: *Logic Programming Paradigm. In Undergraduate topics in computer science*, Cham, Springer International Publishing, 2023

Hunt, John: *A Beginners Guide to Python 3 Programming, Python Properties*, Cham, Springer International Publishing, 2023

Thomas, Iona, Aranega, Vincent, Ducasse, Stephane, Polito, Guillermo & Tesone, Pablo: *A VM-Agnostic and Backwards Compatible Protected Modifier for Dynamically-Typed Languages*, (Scientific Paper: The Art, Science, and Engineering of Programming, Vol. 8, Issue 1, Article 2) 2023, https://www.researchgate.net/publication/371758398_A_VM-Agnostic_and_Backwards_Compatible_Protected_Modifier_for_Dynamically-Typed_Languages