Schrödinger programmiert Python

Das etwas andere Fachbuch

DIE LESEPROBE

Python,
erste schnelle
Schritte

—EINS—

Schrödinger startet durch – mit Python!

Die ersten Schritte in Python sind schnell gemacht, dabei geht es nicht nur um die Syntax – du wirst auch gleich den Zen of Python kennenlernen, in dem (nicht ganz bierernst) ein paar Regeln und Empfehlungen festgelegt sind. Und mit ein paar Variablen und dem Befehl »print« legst du gleich schon so richtig los.

Sag mal, Schrödinger, …

… war da nicht noch was zu tun?

Hast du nicht deiner Freundin versprochen, ihr bei der Buchhaltung zu helfen? Für den Kohl- und Dinkelversand? Wolltest du ihr nicht ein Kassensystem für den Flohmarkt schreiben? Der Flohmarkt findet nächstes Wochenende statt!

Jep, da war was …

Und was ist mit deinen Kumpels vom Fußballverein, deren Vereinsdaten auf Vordermann gebracht werden sollten? In ein paar Wochen ist das große Fußballturnier mit dem beliebten Hornberger Elfmeterschießen.
Du wolltest doch ein Auswertungsprogramm schreiben? Mit einer Verwaltung der Schützen und einer aktuellen Bestenliste …

Da war definitiv was …

Und sag mal, … hattest du nicht deinem Bruder versprochen, ihm bei der Aufbereitung der Bilder und Notizen seiner letzten Amazonas-Expedition zu helfen? Du weißt schon, um die Funde zu erfassen und als Katalog aufzubereiten. Wollte er nicht in zwei Wochen wieder hier sein und loslegen?

Okay, okay, ich fang ja schon an!

Ähm, Stephan?

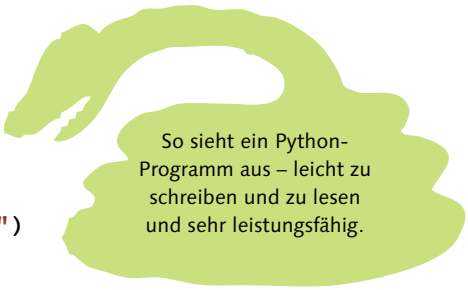
Ja?

Die Programmiersprache …

Schon gut. Ich zeige dir …

Die Programmiersprache Python!

```
eingabe = input("Dein Name?")
if eingabe == "Schrödinger":
    print("Schön, dass du da bist! Ich bin Python.")
```



So sieht ein Python-Programm aus – leicht zu schreiben und zu lesen und sehr leistungsfähig.

Es gibt viele Programmiersprachen, gute, ausgereifte Programmiersprachen, die sich seit vielen Jahren und auf vielen Gebieten bewährt haben.

Aber … ich zeige dir mit Python eine Sprache, die du **rasch** erlernen kannst. Sie ist **einfach** zu schreiben und genauso einfach zu lesen, sodass du schon als Anfänger gut lesbare Programme schreibst, die nicht in einem riesigen Tohuwabohu enden. Es ist eine Sprache, in der sich auch Entwickler anderer Sprachen nach kurzer Einarbeitungszeit zurechtfinden, ohne lange die syntaktische Schulbank drücken zu müssen.

Es ist eine Programmiersprache, die **universell** einsetzbar ist und die nicht nur für das Web oder ganz bestimmte Anwendungsfälle gedacht ist – **plattformübergreifend** für Windows, Linux und Mac verfügbar.

klingt gut.

Objektorientiert und funktional ist Python auch. Und außerdem kommt Python ohne die ganzen Klammern (besonders die geschweiften, die so schlecht zu tippen sind) und ohne Semikolon am Ende jeder Anweisung aus.

Python hat alles Wichtige dabei, also quasi »Batteries included«. Python ist leicht zu **erweitern** – durch Module, die du selbst schreiben kannst, genauso wie durch professionelle Frameworks und fertige Bibliotheken, und das für alle erdenklichen Bereiche wie Bildbearbeitung, numerische Analysen, künstliche Intelligenz oder Datenvisualisierung – eben voll und ganz **wissenschaftlich**.

Mit Python kannst du **schnell** entwickeln – sogar sehr schnell. Trotzdem ist Python nicht etwa auf kleine, einfache Programme beschränkt.

[Zettel]
Python ist eine Sprache, die auch die **Big Player** verwenden, wie Google, Microsoft oder Amazon. Und Python findet in **Wissenschaft** und **Forschung** Anerkennung.

[Hintergrundinfo]
Python wurde Anfang der 1990er Jahre von dem niederländischen Entwickler **Guido van Rossum** erfunden. Er arbeitete damals an einem Forschungsinstitut für Mathematik und Informatik. Vor Weihnachten hatte er ein paar Tage Zeit und suchte nach einem Nachfolger der dortigen Lehrsprache namens ABC. Diese Programmiersprache war für den Unterricht und fürs Prototyping vorgesehen, hatte aber zahlreiche Einschränkungen und ließ sich kaum erweitern. Da er keine geeignete Sprache fand, schrieb er kurzerhand die erste Version der Programmiersprache Python. Das war der Beginn der steilen Karriere einer bemerkenswerten Programmiersprache.

Das Zen of Python und die Sache mit den PEPs

Was macht Python denn nun so besonders?

Nun, folgende Gründe:

- Python hat das »**Zen of Python**«, 20 fast magische Regeln, die eine große Rolle in Python spielen und den »**Geist« (und den Humor) von Python** widerspiegeln – alles festgehalten im Python Developers Guide, in 20 Regeln namens PEP 20.

PEP was?

[Hintergrundinfo]
PEP steht für **Python Enhancement Proposals**, also Verbesserungsvorschläge zu Python. Die bekanntesten sind eben **PEP 20**, das Zen of Python, und **PEP 8**, der Styleguide, mit dem dein Code noch besser, stylischer und professioneller wird. Die PEP 20 findest du übrigens in Anhang C dieses Buches aufgelistet.

- Python ist **weitverbreitet** und hat eine große, aktive Community, von der Python weiterentwickelt wird.
- Python ist leicht zu **erlernen**, da es nur wenige Befehle kennt. Aber keine Sorge, du kannst **alles** damit machen, oft sogar schneller, als du es vielleicht mit anderen Sprachen machen könntest.
- Python ist leicht zu **lesen**, da es auf einige Zeichen verzichtet, die »typisch« für Programmiersprachen, aber eigentlich doch unpraktisch sind: Python kennt zum Beispiel für normale Anweisungen keine geschweiften Klammern (für die man sich auf einer deutschen Tastatur die Finger verbiegen muss) und auch kein »schickes« Semikolon am Ende jeder Anweisung.
- Der Python-Code von Anfängern und Profis ist gleichermaßen gut zu lesen, denn Einrückungen von zusammengehörigen Codeteilen sind fester Bestandteil der Sprache – und korrekt eingerückter Code ist immer besser lesbar.
- Bei der Verwendung von Variablen muss kein Typ (Text, Zahl oder was auch immer) angegeben werden. Python nutzt dafür die sogenannte **dynamische Typisierung** und kümmert sich um alles Notwendige. Ganz praktisch: Eine 1 sieht aus wie eine Zahl, also wird es wohl auch eine Zahl sein, und du sparst dir einiges an Schreibarbeit und Festlegungen. Das ist in Python weniger gefährlich als in manchen anderen Sprachen, denn Python passt genau auf, dass die Typen unterschiedlicher Variablen zueinanderpassen – oder ausdrücklich passend gemacht werden.
- Python hat viele interessante Pakete, Erweiterungen und Bibliotheken.
  Für fast alle Bereiche gibt es fertigen Code, den du ganz einfach verwenden kannst.

Und das sind nur ein paar Punkte, die Python ausmachen.
Mehr – und vor allem die oben genannten 19 Punkte des 20 Punkte umfassenden Zen of Python – wirst du in diesem Buch kennenlernen. Jetzt gleich wirst du deine ersten Schritte mit Python machen …

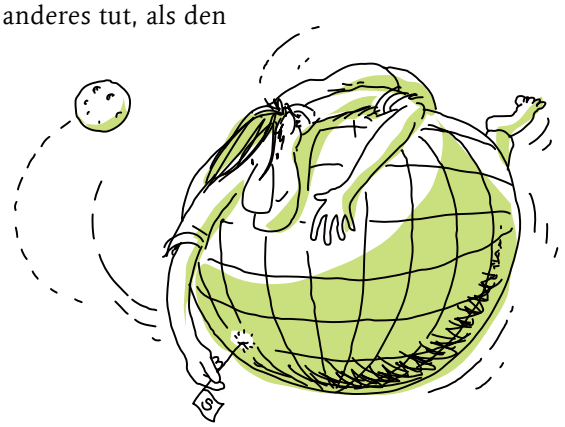
Moment: 19? 20?!!!

Es gibt laut PEP 20 tatsächlich 20 Regeln (bzw. Empfehlungen). Niedergeschrieben sind aber nur 19. Es gibt zahlreiche Diskussionen im Internet, welches wohl der 20. Grundsatz ist und warum er in dieser Auflistung fehlt. Vermutlich ist das dem schrägen Humor geschuldet, der von der namengebenden Komikertruppe Monty Python auf die Programmiersprache abgefärbt hat.

[Hintergrundinfo]
Auch wenn zwei Schlangen das Logo zieren (was ja wirklich sehr gut aussieht), kommt der Name tatsächlich von der legendären britischen Comedy-Truppe Monty Python, die in den 1970er Jahren das Fernsehprogramm und sogar die Kinos mit ihrem unnachahmlichen britischen Humor heimsuchte.

Python, ein erstes »Hallo Welt«

Wie wäre es mit »Hallo Welt«? Das ist der Klassiker, wenn du eine neue Sprache lernst: Du schreibst als Erstes ein Programm, das nichts anderes tut, als den Text **`"Hallo Welt"`** auszugeben.



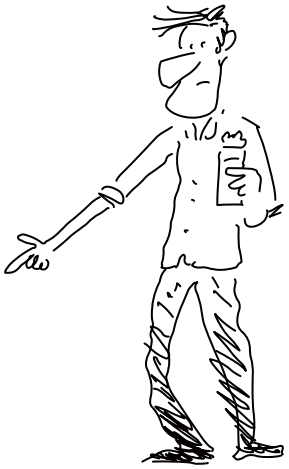
Nichts leichter als das:

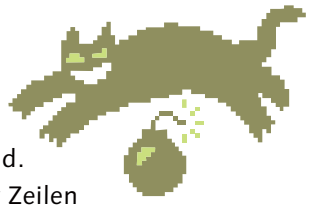
```
mein_text = "Hallo Welt"
print(mein_text)
```

Natürlich, um es gleich vorwegzunehmen: Ein noch einfacheres

```
print("Hallo Welt")
```

würde genauso funktionieren. Aber ich möchte ja, dass du gleich etwas lernst. Also benutze eine **Variable** namens **`mein_text`** und weise ihr einen Wert zu – natürlich den Text **`"Hallo Welt"`**. Und diese Variable gibst du dann mit dem Befehl **`print`** aus.





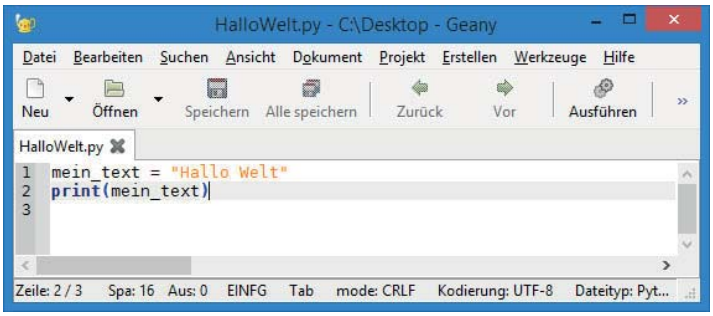
[Achtung]
Python-Code beginnt **ganz links** am Rand.
Es sind **keine Leerzeichen** zu Beginn der Zeilen erlaubt! Einrückungen werden **ausschließlich** verwendet, um zusammengehörende Programmteile zu kennzeichnen (indem sie eben gleich eingerückt werden).

Äh, und wie oder wo führe ich das jetzt aus?

Es gibt **zwei Möglichkeiten**, Python-Code auszuführen:

- **Kurzen Code** und sogar einfache **Berechnungen** kannst du direkt auf der **Kommandozeile** bzw. in der **Python-Shell** ausführen. Das ist ganz praktisch, um mal eben schnell etwas auszuprobieren – oder wenn du etwas berechnen möchtest, aber keinen Taschenrechner zur Hand hast.
- **Mehrzeilige Programme** solltest du in einer **Datei speichern** und dann als richtiges Programm ausführen. Das ist die Arbeitsweise, die wir in erster Linie verwenden wollen. Schließlich willst du ja richtige Programme schreiben, die du speichern und verändern kannst.

Du schreibst den Programmcode also in die Kommandozeile oder mit dem **Editor** deiner Wahl und **speicherst** ihn unter einem beliebigen Namen, aber mit der Endung **.py** ab.



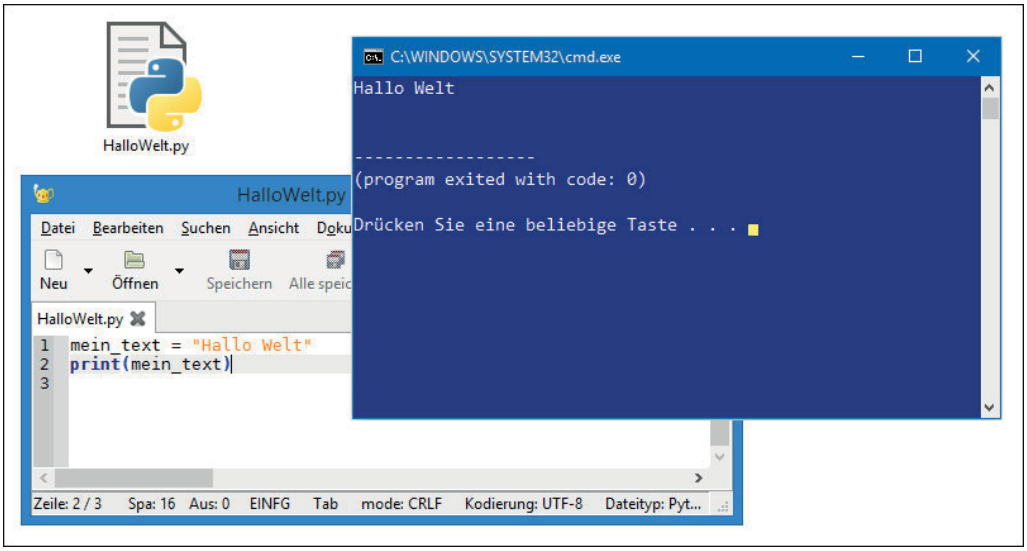
Nur noch schnell in einen Editor eingeben und unter einem fast beliebigen Namen mit der Endung ».py« speichern

Dann noch ein **Doppelklick** auf diese neue Datei, und schon öffnet sich ein Programmfenster, in dem dein Text ausgegeben wird. Wenn du einen speziellen **Editor** wie Thonny oder Mu hast (oder eine Entwicklungsumgebung), kannst du dein Programm auch direkt darüber starten. Bei dem Editor Geany beispielsweise genügt es, auf den Button **Ausführen** zu klicken. Verwendest du IDLE, die Integrated Development and Learning Environment, dann genügt es, die Taste **F5** zu drücken, um dein Programm zu starten.

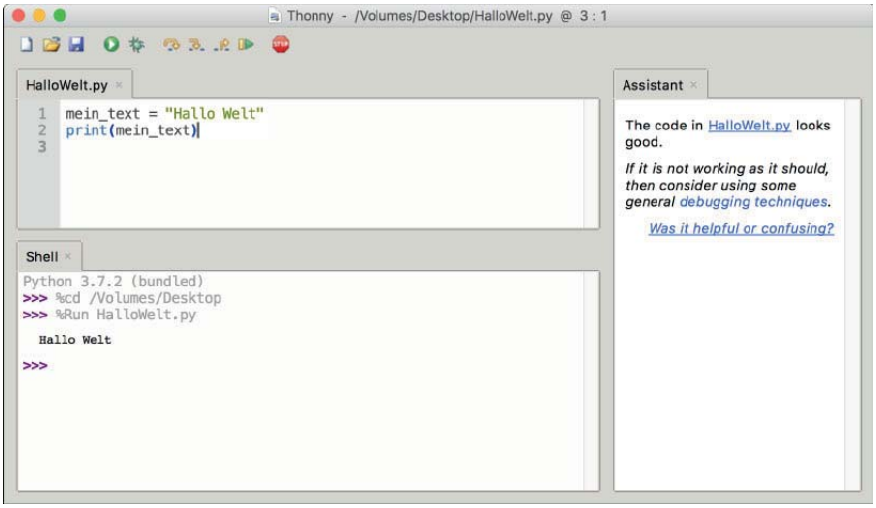
Du hast noch keinen Editor?

Oder hast du kein Python, um die Programmdatei zu starten oder für die Kommandozeile? Dann ist **jetzt** der richtige Zeitpunkt, genau das **in den verlorenen Kapiteln** im **Anhang** nachzulesen. Dort erfährst du mehr darüber, woher du Python bekommst, wie du es installierst und wie du deine Programme schreiben und starten kannst.

Wir sehen uns hier wieder.
Alles klar und bereit? Dann weiter …



So kann dann das erste »Hallo Welt« aussehen, hier auf der Kommandozeile von Windows. »code: 0« heißt so viel wie »kein Fehler«.



Und so sieht unser »Hallo Welt« mit dem Editor Thonny aus. Auch nicht schlecht …

Schauen wir uns mal genauer an, was da gemacht wurde:

Ein Python-Programm (also geschriebener Quelltext im Editor) **fängt einfach an**. Leerzeilen spielen dabei erst mal überhaupt **keine Rolle**. Du schreibst die Anweisungen und Befehle **Zeile für Zeile** in eine Datei. Es gibt keine Klassen oder andere Konstrukte, die du zuerst als »Rahmen« für das Programm definieren müsstest. Auch der **Name** der Datei, in der du dein Programm speicherst, folgt nur den Regeln des (hoffentlich) gesunden Menschenverstandes. Wichtig ist nur, dass dein Betriebssystem den Dateinamen akzeptiert.

*1 Ein **gültiger Name** einer Variablen beginnt mit einem Buchstaben oder einem Unterstrich. Danach sind Buchstaben, Zahlen und auch wieder Unterstriche erlaubt. Auch Umlaute kannst du problemlos verwenden.

*2 Eine **Zuweisung** geschieht mit einem einfachen **=**. Rechts von dem **=** kann ein **Literal** (also ein echter, fester Wert), eine andere **Variable** mit einem Wert oder eine ganze **Operation** stehen, die einen Wert als Ergebnis erzeugt. Dieser Wert wird dann in der Variablen gespeichert.

```
mein_text*1 =*2 "Hallo Welt"*3*4
```

*3 Das ist ein **Literal**, hier in Form eines Textes, in der Programmierung **String** genannt. Gut zu erkennen ist ein String, da einfache **oder** doppelte Anführungszeichen den Text umschließen.

*4 Das ist das **Ende** dieser Anweisung – natürlich noch nicht das Ende des Programms. In der nächsten Zeile geht es ja weiter. Sicher ist dir aufgefallen, dass hier am Ende **gar nichts** steht, kein Semikolon und kein anderes Zeichen. Python verzichtet auf ein abschließendes Zeichen.

Was ist denn der Unterschied zwischen einfachen und doppelten Anführungszeichen?

Es gibt keinen, beide sind **gleichberechtigt**! Der Vorteil ist, dass du das jeweils andere Anführungszeichen dann ohne Probleme **innerhalb** des Textes verwenden kannst: `"Hallo 'liebe' Welt"` oder `'Hallo "liebe" Welt'` kannst du so problemlos schreiben.

*1 Das klassische **print** sorgt dafür, dass etwas ausgegeben wird. Was? Das steht nicht in den Sternen, sondern innerhalb der runden Klammern, …

*2 … die du hier siehst. Die Klammern werden geöffnet und (klar) wieder geschlossen.

```
print*1(*2mein_text*3) *4
```

*3 Alles, was in den Klammern steht, wird ausgegeben. Das könnte ein Literal, eine Operation oder wie hier eine Variable sein.

*4 Auch hier wieder am Ende: kein Semikolon, keine End-Anweisung, keine geschweifte Klammer, weder am Ende der Zeile noch in der Zeile darunter.

Fertig! Dein (aller-)erstes Python-Programm

Fingerübungen mit »print«

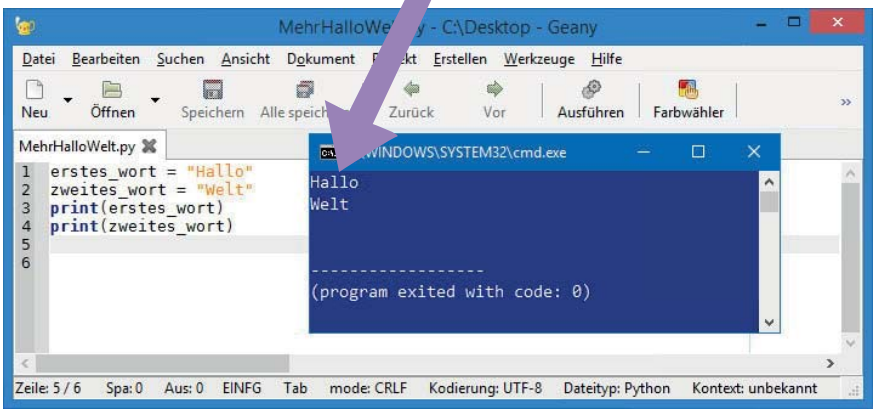
Machen wir einfach ein paar Übungen dazu. So bekommst du etwas Praxis im Umgang mit Variablen und wirst sehen, wie flexibel der Befehl **`print`** ist.

[Einfache Aufgabe]

Verwende bitte für jedes Wort von **`"Hallo Welt"`** eine eigene Variable und gib sie dann mit **`print`** aus.

Lösung:

```
erstes_wort = "Hallo"
zweites_wort = "Welt"
print(erstes_wort)
print(zweites_wort)
```



Ausgegeben wird alles – aber gleich in zwei Zeilen!?

Jede Ausgabe steht in einer eigenen Zeile. Das könnten wir natürlich zur gewünschten Lösung erklären, aber so einfach wollen wir es uns doch nicht machen. **Zumal der Befehl `print` einiges zu bieten hat!**

[Notiz]

Jedes **`print`** schreibt seinen Inhalt in eine neue Zeile.

Hallo Welt in (einzeiligen) Variationen

1. Du fügst die beiden Wörter in einer neuen Variablen zusammen und gibst den Inhalt dieser neuen Variablen dann mit einem **`print`**-Befehl aus:

*1 Du kannst **mehrere Zuweisungen** in **einer Zeile** zusammenfassen. Du hast auf der linken Seite des **=** mehrere Variablen und genauso viele Werte auf der rechten Seite. Die Variablen und die Werte sind jeweils durch ein Komma getrennt.

*2 Unsere neue Variable namens **`ergebnis`** gilt für die beiden zusammengefügten Wörter.

```
erstes_wort, zweites_wort*1 = "Hallo", "Welt"
ergebnis*2 = erstes_wort +*3 " "*4 + zweites_wort
print(ergebnis)
```

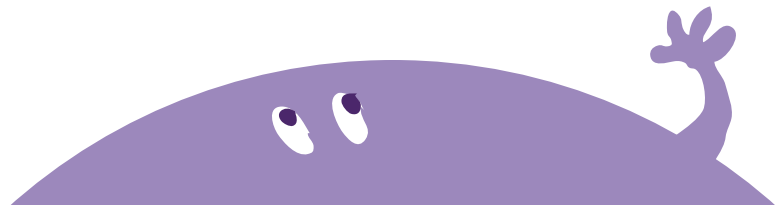
*3 Strings werden in Python ganz einfach mit einem **+** aneinandergehängt.

*4 Das **Leerzeichen** solltest du auch einfügen: Ohne Leerzeichen käme **`"HalloWelt"`** heraus.

Python weiß, dass ein **+** zwischen Strings kein mathematisches + ist, sondern Texte zu einem langen Text **zusammenfügt**. Dabei spielt es keine Rolle, ob die Texte als Literale oder Variablen angegeben sind. Wie du in unserem Beispiel siehst, können Variablen und Literale (hier unser einsames Leerzeichen) problemlos miteinander gemischt werden.

2. Wir geben beide Variablen zusammen aus. Gemeinsam im Befehl **`print`** – ohne alles erst in einer Variablen zu speichern. Auch das ist eine durchaus gängige Praxis:

```
erstes_wort, zweites_wort = "Hallo", "Welt"
print(erstes_wort + " " + zweites_wort)
```



3. Oder wir machen es nach Python-Art:
Python ist eine sehr **praktische** Programmiersprache. Die **`print`**-Funktion ist so **flexibel**, dass du es schneller und einfacher machen kannst als mit dem **+**. Eben auf the Python way: Du kannst **`print`** mehrere Werte übergeben, die du einfach durch ein Komma trennst.

```
print(erstes_wort,zweites_wort)
```

Ja, aber hast du nicht das Leerzeichen vergessen?

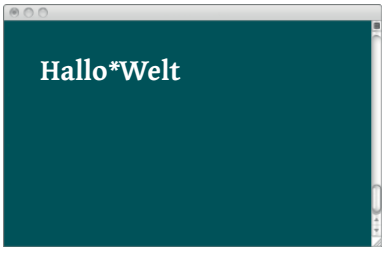
Nein, **`print`** fügt **automatisch** zwischen alle angegebenen Werte ein Leerzeichen ein. Verantwortlich dafür ist ein **optionaler Parameter** namens **`sep`**. Das steht für **Separator**, also Trennzeichen. Diesem Parameter kannst du einen beliebigen String zuweisen, der dann zwischen alle angegebenen Werte eingefügt wird. Verwendest du **`sep`** nicht (denn dieser Parameter ist ja optional), dann wird automatisch ein Leerzeichen zwischen alle Elemente gesetzt. Probier es mal mit einem * aus:

*1 Leerzeichen in den Anweisungen und **zwischen** den auszugebenden Elementen dienen nur der **Übersichtlichkeit** und haben **keinen** weiteren Einfluss.

```
print(erstes_wort, *1zweites_wort, *1sep='*'*2)
```

*2 Was **`sep`** zugewiesen wurde, wird zwischen jedes Element bzw. jeden Wert geschrieben, den du im **`print`** angegeben hast. Du kannst **`sep`** einen beliebigen (und auch beliebig langen) Text zuweisen, sogar einen leeren String, also **`""`**.

Die Ausgabe:



[Code bearbeiten]
Probiere es einfach mal mit verschiedenen Werten anstelle des Sternchens selbst aus. Nun noch alles in einer Python-Datei deiner Wahl **speichern** und **ausführen** lassen.

4. Das ist aber noch gar nicht alles. Die **`print`**-Funktion kennt noch einen weiteren optionalen **Parameter** namens **`end`**. Normalerweise schreibst du mit jedem **`print`** eine neue Zeile. Mithilfe von **`end`** kannst du das ändern und anstelle des Zeilenumbruchs ein anderes Zeichen angeben:

```
erstes_wort, zweites_wort = "Hallo", "Welt"
print(erstes_wort, end=" ")  *1
print(zweites_wort)  *2
```

*1 Hier wird der Zeilenumbruch »umgebogen« zu einem **Leerzeichen**. Genau das, was wir brauchen.

*2 Hier müssen wir nichts ändern. Der standardmäßige Zeilenumbruch darf hier bleiben. Anders wäre es, wenn wir weitere Worte mit einem eigenen **`print`** in die gleiche Zeile schreiben wollten.

Die Parameter **`sep`** und **`end`** müssen für jedes **`print`** explizit angegeben werden – zumindest wenn du nicht das normale Verhalten von **`print`** haben möchtest: Ohne diese Angaben hat das jeweilige **`print`** nämlich wieder die (unsichtbaren) Standardwerte **`sep=' '`** und **`end='\n'`**.

[Notiz]
Das **`\n`** steht für eine neue Zeile. Auf diese etwas seltsame Art, beginnend mit einem **`\`**, können sogenannte **Steuerzeichen** geschrieben werden. **`\n`** steht dabei für **new line**, also für einen Zeilenumbruch. **`\t`** ist ein Tabulator, **`\b`** ein Backspace (es geht also wieder eine Stelle nach links).

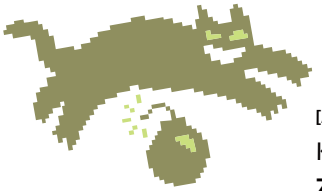
Puh, wie Möglichkeiten muss für »Hallo Welt«! Und welche soll ich jetzt nehmen?

Nun, alle Möglichkeiten machen Sinn. Meist ergibt es sich aus der konkreten Aufgabe, welche Methode am sinnvollsten ist. Wichtig ist, dass du weißt, dass es (eigentlich immer) unterschiedliche Möglichkeiten gibt, ans Ziel zu kommen. Such dir einfach das Passendste heraus.

Wir müssen reden: Du und deine Variablen

Variablen spielen in der Programmierung – und damit natürlich auch in Python – eine große Rolle. Sie sind so etwas wie Speicher, die einen Wert aufnehmen können. Wie du bereits gesehen hast, ist das Arbeiten mit Variablen recht einfach: Variablen **entstehen**, indem du ihnen einen Wert **zuweist**. Woher dieser Wert kommt – ein Literal (also ein fester Wert wie ein Text oder eine Zahl), eine Berechnung oder irgendeine andere Operation –, spielt erst mal keine Rolle.

```
neue_variable = "Ich bin ein Literal"
andere_variable = 42 * 2 / ( 0.5 * 4 )
```



[Achtung]
Keine Variable kann **ohne Zuweisung** existieren!

Du darfst **keine** Variable verwenden, **ohne** ihr zuvor einen Wert zugewiesen zu haben.
Also, was ist hier richtig, und was ist falsch?

```
spam = eggs *1 * 2 X
```

*1 Genau! Python mag die (noch) unbekannte und leere Variable namens **eggs** überhaupt nicht.

In diesem Fall wäre **spam** eine gültige Variable, nur **eggs** ist leider noch unbekannt (und hat auch keinen Wert) und so kommt es zu einem **Fehler**.

```
C:\WINDOWS\SYSTEM32\cmd.exe
Traceback (most recent call last):
  File "meinProgramm.py", line 1, in <module>
    spam = eggs * 2
NameError: name 'eggs' is not defined


------------------
(program exited with code: 1)
```

Die Fehlermeldung mag erst etwas kryptisch erscheinen, gibt aber einen eindeutigen Hinweis, nur leider auf Englisch.

Erst wenn du vorher **eggs** einen Wert zuweist, wird Python dein Programm akzeptieren:

```
eggs = 1
spam = eggs * 2
```

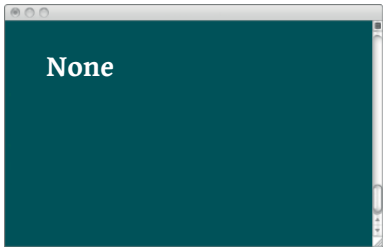
Genauso wenig funktioniert übrigens ein **print** mit einer Variablen, der noch kein Wert zugewiesen wurde:

Erst **nach** einer Zuweisung eines Wertes funktioniert das:

```
eggs = "So wird das was"
print(eggs)
```

Oder anders gesagt: Bevor eine Variable in einer Funktion wie **print** oder einer Operation wie **eggs * spam** verwendet werden darf, muss sie erst mal einen Wert erhalten haben. Dabei sind auch **0** oder **''** (zwei Anführungszeichen **ohne** Inhalt), also Texte ohne Inhalt, gültige Werte. Es gibt übrigens auch einen speziellen Wert **None**, der für ein voll pythonisch korrektes »**gar nichts**« steht:

```
eggs = None*1
print(eggs)
```



Was hat es eigentlich mit **spam** und **eggs** auf sich?

[Hintergrundinfo]

In Python ist es guter Ton, **`spam`** und **`eggs`** zu verwenden, wenn man für ein **Codebeispiel** beliebige Variablennamen braucht – sogenannte metasyntaktische Variablen.

`spam` und **`eggs`** gehen auf einen Sketch der englischen Comedy-Gruppe Monty Python zurück: Darin versucht ein Gast in einem Restaurant ein Gericht ohne Spam zu bestellen. Spam ist salziges Frühstücksfleisch in eckigen Dosen, die sich kaum ohne Verletzung öffnen lassen.

Wie zu erwarten, gibt es nur Gerichte mit Spam – und zufällig anwesende Wikinger (!) singen in dem Sketch regelmäßig einen Lobgesang auf Spam.

Und jetzt rate mal, wo das Wort Spam für unerwünschte Mails herkommt …

[Achtung]

Variablen **entstehen**, indem ihnen ein Wert zugewiesen wird. Solange eine Variable keinen Wert hat, kannst du sie auch nicht verwenden. Das würde einen **Fehler** verursachen.

[Hintergrundinfo]

Durch die erste Zuweisung eines Wertes, die sogenannte **Initialisierung**, wird eine Variable **deklariert** – sie wird vereinfacht ausgedrückt dem System bekannt gemacht. Ab diesem Zeitpunkt kannst du sie verwenden. Die Initialisierung und die Deklaration sind wichtige Konzepte in praktisch allen Programmiersprachen.

Und du hast ja schon gesehen, dass du sogar mehrere Zuweisungen **zusammenfassen** kannst:

```
spam, eggs, brot = "lecker", "gekocht", "geschnitten"
kleine_zahl, grosse_zahl = 100, 1
```

Jede Variable bekommt der Reihenfolge nach einen Wert zugewiesen.
Die Anzahl der Variablen **links** und die Anzahl der zugewiesenen Werte auf der **rechten** Seite müssen natürlich genau passen, sonst gibt es einen Fehler.

Moment, du hast die Werte in der zweiten Zeile vertauscht!

Du hast recht! Aber mit Python sind die Werte zweier Variablen **schnell getauscht**:

```
kleine_zahl, grosse_zahl = 100, 1
kleine_zahl, grosse_zahl = grosse_zahl, kleine_zahl
```

Das geht ganz einfach, denn du kannst einer Variablen problemlos den Wert einer anderen Variablen zuweisen.

In der Programmierung müssen oft die Werte zwischen Variablen getauscht werden, zum Beispiel wenn große und kleine Werte verglichen und dann zum Sortieren getauscht werden müssen. Mit Python geht das schnell und einfach mit einer einzigen Zeile Code!

Übrigens… Variablennamen werden nach **Groß- und Kleinschreibung** unterschieden, `mein_Text` ist also eine ganz andere Variable als `Mein_text` oder `mEiN_tExT`. `eggs` ist eine andere Variable als `Eggs`. Wenn du dich beim Schreiben von Variablennamen mal vertippen solltest, wird dich Python bereits beim Programmstart mit einer Fehlermeldung darauf hinweisen, dass da eine unbekannte oder falsch geschriebene Variable im Quelltext sitzt.

[Hintergrundinfo]

Auch wenn Python eher niederländisch-locker ist, es gibt eine Namenskonvention nach PEP 8: Alle Namen von Variablen und Funktionen (die kommen noch) sollten **klein mit Unterstrich(en)** geschrieben werden. Einzelne Worte in den Namen werden dabei zur besseren Lesbarkeit mit Unterstrichen getrennt. Das ist **lower_case_with_underscores**.

[Begriffsdefinition]

Namenskonventionen sind Regeln bzw. Vorschläge, wie du Namen von Variablen oder Funktionen schreiben solltest. Konventionen deshalb, weil es nur Empfehlungen sind. Wenn du dich (vielleicht aus gutem Grund) nicht daran hältst, gibt es keinen Fehler und auch keine Warnung.

Variablen – was geht? Und was ist voll krass korrekt?

Hier sind erst mal ein paar **gültige** Variablennamen. Schau sie dir bitte mal an:

```
_wert = "Hallo"*1
wortl = "Schrödinger"*2
wörtlich = "wie"*3
wort_drei*1 = "geht's"
```

*1 Der Unterstrich ist das **einzige, erlaubte Sonderzeichen**, egal, ob am Anfang, innerhalb des Namens oder am Ende.

*2 **Zahlen** sind natürlich auch im Namen erlaubt (nur eben nicht an erster Stelle).

*3 **Umlaute** sind auch kein Problem.

[Einfache Aufgabe]

Lass dir diese Variablen mit einem **`print`** ausgeben, sodass zwischen jedem Wort ein Bindestrich »-« steht. Am Ende soll ein Fragezeichen stehen und trotzdem noch ein Zeilenumbruch gemacht werden. Spicken ist wie immer erlaubt …

Lösung:

```
print(_wert, wortl, wörtlich, wort_drei, sep="-", end="?\n")
```

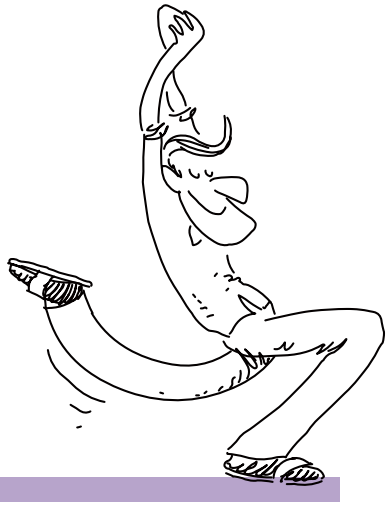
Und zum Schluss noch ein passendes **`end`** dazu, dem du ein **Fragezeichen** mit dem geforderten **Zeilenumbruch** zuweist:

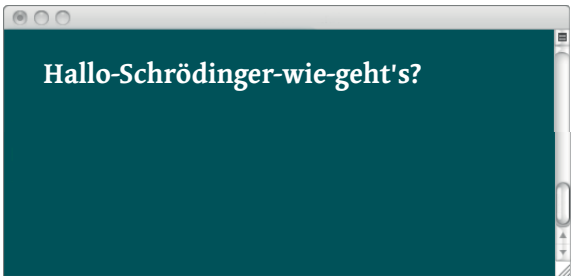
```
print(_wert, wortl, wörtlich, wort_drei, sep="-")
```

Danach »baust« du mit dem Parameter **`sep`** noch den Bindestrich zwischen den Wörtern ein:

```
print(_wert, wortl, wörtlich, wort_drei)
```

Erst mal ganz einfach: die Ausgabe an sich.



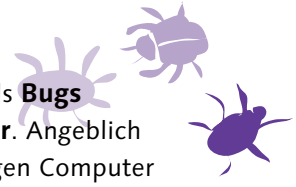


Super, das war doch gar nicht so schwer.

Und hier auf besonderen Wunsch der Käferfraktion ein paar Fehler bzw. Variablennamen, die du **nicht** nehmen darfst, weil sie **falsch** sind!

[Notiz]

Fehler werden in der Programmierung gerne als **Bugs** bezeichnet. Ein Bug ist im Englischen ein **Käfer**. Angeblich rührt das von der Zeit der ersten, damals riesigen Computer her. Eine kleine Motte hatte in einem Computer einen Schalter außer Gefecht gesetzt, was zu unerklärlichen Fehlern im Programm führte.



Agent Schrödinger, übernehmen Sie!

```
*1$wort = "Vorsicht"  X
*2lwort = "Schrödinger"  X
*3wort_zwei "so geht"  X
*4Wort-drei = "es NICHT"  X
```

*1 Das Dollarzeichen ist ein unerlaubtes Sonderzeichen, und zwar egal, ob am Anfang wie hier oder innerhalb des Namens.

*2 Am Anfang darf keine Zahl stehen.

*3 …? Moment! Der Name der Variablen ist richtig – nur das **ist-Zeichen** bei der Zuordnung fehlt!

*4 Der Bindestrich ist ein unerlaubtes Sonderzeichen im Namen. Außerdem entspräche der Name nicht der Namenskonvention alles Kleinzuschreiben (was aber kein Fehler wäre).

Schrödinger, schau dir mal folgende Aufgaben an:

[Einfache Aufgabe]
Da der Fehlerteufel in Käferform an vielen Stellen zuschlagen kann: Was ist **richtig**? Was ist **falsch**? Und warum? Jeder Zettel steht übrigens für sich allein.

1.

```
spam = "Irgendwas mit " + egg
```

2.

```
spam = spam + 'Brot'
```

3.

```
spam = 'Ist das "etwa" richtig?'
```

4.

```
egg = "Die Fälschung ist 'echt' falsch"
```

5.

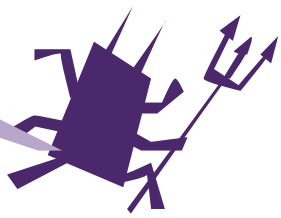
```
egg = 'Ich bin ein Literal"
```

6.

```
egg = "Schrödinger"
spam = 'Hallo' + Egg
```

7.

```
spam = "Ja" * 3
```



Lösung:

1. Falsch. Die Variable `egg` ist unbekannt, sie hat noch keinen Wert.

2. Falsch. `spam` ist noch unbekannt und darf noch nicht Teil einer Operation (auf der rechten Seite des `=`) sein. Das geht natürlich nicht.

3. Stimmt! Wenn der String durch einfache Anführungszeichen umschlossen ist, kannst du die doppelten Anführungszeichen wie jedes andere Zeichen im String verwenden – übrigens als Zeichen im Text nicht nur paarweise, sondern beliebig oft.

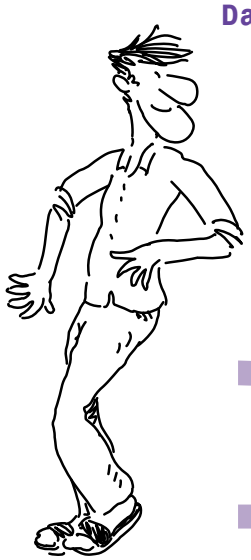
4. Stimmt auch! Der String ist von **doppelten** Anführungszeichen umschlossen. Also kannst du im String die **einfachen** Anführungszeichen verwenden – wieder beliebig oft oder auch nur einmal.

5. Falsch, denn ein String muss immer von der **gleichen Art von Anführungszeichen** umschlossen werden. Hier beginnt der String mit einem einfachen und endet mit einem doppelten Anführungszeichen. Umgekehrt wäre es natürlich genauso falsch.

6. Falsch! `egg` ist eine ganz andere Variable als `Egg` – Python unterscheidet ja zwischen Groß- und Kleinschreibung.

7. Das funktioniert tatsächlich! Das Ergebnis ist dreimal der angegebene String aneinandergefügt: »JaJaJa«. Strings können ja mit einer Integer-Zahl (und **nur** damit) multipliziert werden!

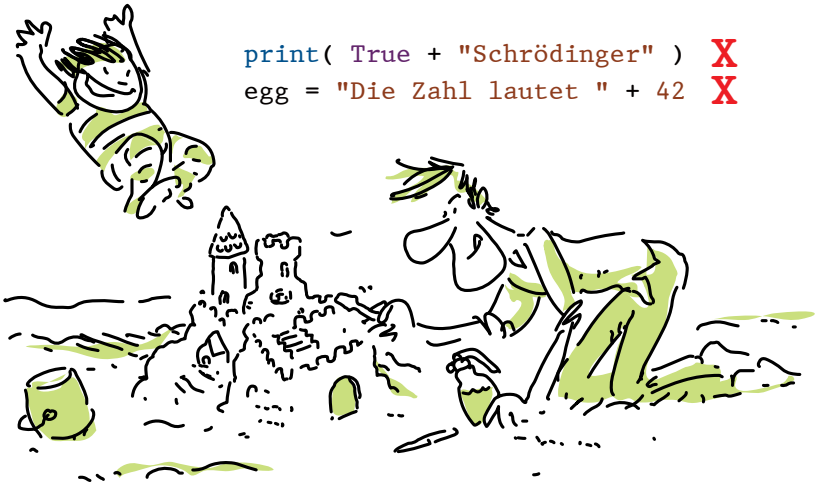
Das war doch gar nicht so schwer!



Ach, Schrödinger. Bevor ich es vergesse, ist hier noch eine wichtige Sache.

Die Sache mit den (Daten-)Typen

Schau dir bitte mal das hier an:



```
print( True + "Schrödinger" )  X
egg = "Die Zahl lautet " + 42  X
```

Wenn es sich nicht ausschließlich um Zahlen handelt, darfst du unterschiedliche Arten von Werten **nicht in Berechnungen bzw. Operationen vermischen**. In der Programmierung werden nämlich Daten nach ihrem **Typ** unterschieden: **Zahlen** und **Strings** kennst du ja schon. Und Zahlen werden sogar noch weiter unterschieden:

- **ganzzahlige Zahlen** wie 1 oder 42 oder 2021, ohne Stellen nach dem Komma
- **Fließkommazahlen** wie 1.9 oder 3.141
- **komplexe Zahlen**, wie du sie vielleicht aus dem Matheunterricht oder dem Studium kennst

Außerdem gibt es noch **boolesche Werte** für wahr, `True`, oder falsch, `False`, in Python. Und dann wirst du noch ein paar andere, gar nicht mal so komische Typen kennenlernen.

[Achtung]
In Python wird, wie in fast allen Programmiersprachen, das **Komma** bei Fließkommazahlen (wie bei 3,14159) als **Punkt** geschrieben: `3.14159`

[Hintergrundinfo]
Nicht umsonst ist eine Fließ**komma**zahl im Englischen eine »floating **point** number«. Und da Englisch die Sprache der Programmierung ist, richten sich Programmiersprachen hier nach ihr.

Für den Computer ist die Frage des Datentyps wichtig: Ein String muss intern anders behandelt und gespeichert werden als ein Wahrheitswert: Ein String kann fast beliebig lang (oder kurz) sein, während ein boolescher Wert nur `True` oder `False` (also wahr oder falsch bzw. intern **0** oder **1**) darstellt.

Dabei können boolesche Werte, Integer (also ganzzahlige Werte) und Fließkommazahlen recht **gut miteinander** in Berechnungen verwendet werden. In Berechnungen mit Zahlen werden `True` zu `1` und `False` zu `0` umgewandelt. Integer wie `1` oder `42` werden in Berechnungen mit Fließkommazahlen wie `3.141` automatisch in Fließkommazahlen `1.0` oder `42.0` umgewandelt. Hier macht Python die Arbeit für dich ganz automatisch – denn hier ist es ziemlich **eindeutig** wie die Werte zu verstehen sind, und es besteht **keine Gefahr**, dass Werte durch eine automatische Umwandlung verändert oder falsch verstanden werden.

Schau dir bitte mal die Berechnung in dem `print` an und sag mir, was dabei herauskommt:

```
print( 42 + 12.0 * False )
```

Also Schrödinger, Python arbeitet nach den normalen Regeln der Mathematik: **Punktrechnung vor Strichrechnung**.

Somit beginnt die Berechnung rechts mit der Multiplikation. Das `False` wird zu einer Fließkommazahl `0.0` umgewandelt, damit es passend zur Fließkommazahl `12.0` ist. Das ergibt `12.0 * 0.0` und als (Teil-)Ergebnis `0.0`. Das Ergebnis `0` bleibt übrigens eine Fließkommazahl als `0.0`.
Dann berechnet Python `42 + 0.0`. Damit es passend wird, macht Python aus der `42` die Fließkommazahl `42.0` und rechnet die beiden Zahlen zusammen.

Das Ergebnis ist 42.0!

Wenn das mit den Zahlen doch so gut funktioniert, warum dürfen dann ein Text und eine Zahl nicht zusammengefügt werden?

Nun, so lange es **eindeutig** ist, wie Werte zu interpretieren (und damit umzuwandeln) sind, so lange übernimmt Python das gerne für dich – ganz **automatisch**. Aus einer `42` wird in einer Operation mit einer Fließkommazahl immer eine `42.0`. Wichtig für Python ist auch, dass keine Informationen dabei verloren gehen. Deshalb findet immer eine automatische Konvertierung zu dem **höherwertigen** Datentyp statt.

Boolean → **Integer** → **Fließkommazahl**

Andersherum geht es **nicht**, zumindest nicht automatisch. Denn wie sollte anders (beispielsweise) aus einer 17.7 eine Integer-Zahl werden? Runden? Aufrunden? Abrunden? **Nein, das ist zu unsicher!** Und wie sollte es erst recht funktionieren, wenn Zahlen und Texte miteinander verknüpft werden sollen?

Wie sagt das Zen of Python?

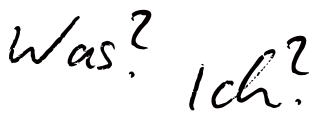
»In the face of ambiguity, refuse the temptation to guess. – Wenn etwas mehrdeutig ist, widerstehe der Versuchung zu raten.«

Es gibt Programmiersprachen, die **versuchen**, selbst zu erkennen, was sinnvoll ist und wohl gemeint sein könnte: Ist also eine `"42"` ein String? Oder ist die Zahl gemeint? Gerade bei einer Addition:

```
text_oder_zahl = "42" + 23X
Das lässt Python gar nicht zu! Es kommt zu einem Fehler!
```

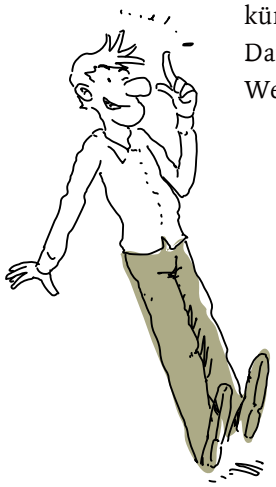
Was würdest du hier als Ergebnis erwarten? Die Zahl `65` als Ergebnis einer Addition oder den zusammengefügten Text `"4223"`? Was ist sinnvoll?

Python hat hier eine ganz eindeutige Antwort: **Das kann niemand wissen – nur der Programmierer, also du!**



Schrödinger, das Zen of Python sagt dir: »Explicit is better than implicit. – Explizit ist besser als implizit.«

Also sagt Python, es ist das Beste, **du** legst ausdrücklich (**explizit**) fest, wie die Werte zu verstehen sind. **Du** musst dich darum kümmern, wie dein Programm mit unterschiedlichen Daten bzw. Datentypen umgeht. **Du** musst explizit festlegen, wie die einzelnen Werte in einer Operation zu verstehen sind!



[Begriffsdefinition]

Explizit bedeutet, du musst es **ausdrücklich** angeben, also in Form eines Befehls oder einer Anweisung wirklich schreiben.

Python **passt genau auf**, dass unpassende Datentypen nicht einfach addiert oder mit anderen Operationen einfach »mal so« zu neuen Werten verarbeitet werden – es sei denn, es ist eindeutig und es gehen keine Informationen dabei verloren. Und von einigen, genau festgelegten **Sonderfällen** einmal abgesehen.

Sonderfälle?

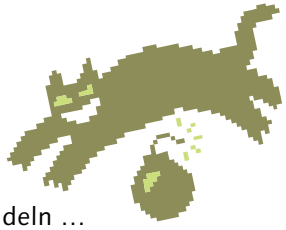
Zum Beispiel die Multiplikation von Strings mit Integer-Zahlen:

```
zeichenkette = "-" * 10
print( zeichenkette )
```

```
----------
```

Du kannst tatsächlich einen String mit einer Zahl multiplizieren. Der String wird dann **vervielfacht**, das ist durchaus ganz praktisch – **`willkommen in der wunderbaren Welt von Python.`**

Ansonsten gilt aber: Erst wenn Daten explizit – also auf deine Anweisung hin – zu einem **gleichen Datentyp** umgewandelt sind, kannst du sie gefahrlos vermischen, addieren oder sonst etwas damit machen.



[Achtung]
Für **Berechnungen** musst du die beteiligten Daten zu **einem gleichen Datentyp** umwandeln …

`… dafür ist es natürlich wichtig zu wissen, welche (Daten-)Typen es so gibt!`

Werfen wir einen raschen Blick auf die (erst mal) **wichtigsten Datentypen**, die du zum Programmieren benötigst.

Diese Datentypen sind für dich da!

- **Strings bzw. Texte**

```
'Hallo' "42"

'10 Vorne' "Schrödinger"
```

Strings kennst du ja. Gekennzeichnet sind sie durch Anführungszeichen am Anfang und am Ende. Wichtig ist, dass eine **Zahl**, die sich **innerhalb** solcher Anführungszeichen befindet, auch ein **String** ist: `"42"` ist also etwas ganz anderes als die Zahl `42`.

- **Integer-Zahlen**

```
42   451   -100   12334578980732789283304203829O3
```

Integer-Zahlen, also Ganzzahlen, können positiv oder negativ sein, haben aber eben keinen Nachkommateil. Eine **Besonderheit** von Python ist, dass Integer-Zahlen praktisch **beliebig groß** werden können.

- **Fließkommazahlen**

```
42.0   -191.0   3.14159   .25   1e2   134.
```

Fließkommazahlen werden mit einem Punkt geschrieben. Aber auch Zahlen in der **Exponentialschreibweise** (wie eben `1e2`, also `100.0`) sind Fließkommazahlen.

- **Boolean, Wahrheitswerte**

```
True False
```

Ist etwas **wahr** oder **falsch**? Das sagen dir die **booleschen** Werte. `True` und `False` müssen übrigens exakt so geschrieben werden: Der erste Buchstabe ist **groß**, der Rest wird **kleingeschrieben**.
Sie dürfen auch **nicht** in Anführungszeichen geschrieben werden – dann wären sie nämlich normale Strings.

Und was mache ich jetzt damit?

Über den richtigen Kamm scheren – Datentypen konvertieren

Python hat Funktionen, mit denen du Werte von **einem Datentyp** in einen **anderen Typ** umwandeln kannst. Vorausgesetzt, das geht von den Werten her überhaupt: Denn man kann nicht jeden Wert sinnvoll in einen anderen Datentyp konvertieren.

`Man kann eben nicht aus jeder Mücke einen Elefanten machen.`

Um beispielsweise für eine Berechnung unpassende Werte in **Zahlen** zu verwandeln, kannst du die Funktion **`int()`** verwenden: Damit wird jeder Wert in einen Integer, also eine ganze Zahl, umgewandelt – zumindest soweit das möglich ist. Aus **`"Haus"`** könnte eben keine Zahl erzeugt werden. Eine Fließkommazahl würde damit in eine Integer-Zahl verwandelt, indem einfach alles nach dem Komma (bzw. Punkt) abgeschnitten würde.

Willst du eine Fließkommazahl erzeugen, hilft dir dabei die Funktion **`float()`**. Um etwas in einen String zu verwandeln, gibt es dafür die Funktion **`str()`**.

[Einfache Aufgabe]

Addiere den String **`"42"`** als Zahl zu der Zahl 23 und weise das einer Variablen zu.

```
eine_zahl = int("42")*1 + 23
```

*1 Mit der Funktion **`int()`** wird ein darin angegebener Wert in eine Integer-Zahl umgewandelt, zumindest wenn der Wert in die andere »Form« als Zahl passt.

[Einfache Aufgabe]

Und jetzt füge die gleichen Werten als Strings zusammen und weise das Ergebnis einer Variablen zu.

```
ein_text = "42" + str(23)*2
```

*2 Die Funktion **`str()`** macht aus dem angegebenen Inhalt einen String. Das Ergebnis ist **`"4223"`** – zwei Strings wurden zusammengefügt und das Ergebnis dann der Variablen zugewiesen.

So klappt es mit dem Nachbarn, der in diesem Fall eben einen anderen Datentyp hat. Allerdings hat das Grenzen, denn aus **`"10 Vorne"`** oder **`"Hausnummer 13"`** kannst du auch mit **`int()`** keine Zahl machen.

Die Konvertierung der Werte funktioniert natürlich nicht nur mit Literalen (wie oben), sondern genauso mit Variablen oder Werten, die aus Operationen entstehen.

[Einfache Aufgabe]

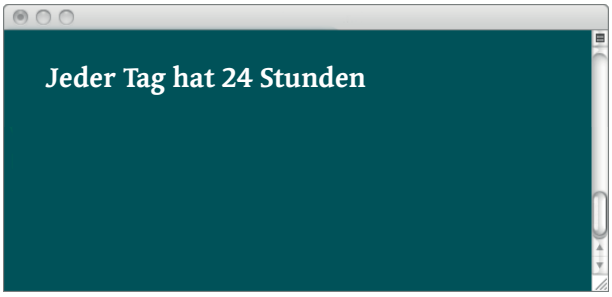
Was wird hier als Ergebnis von **`print`** ausgegeben?

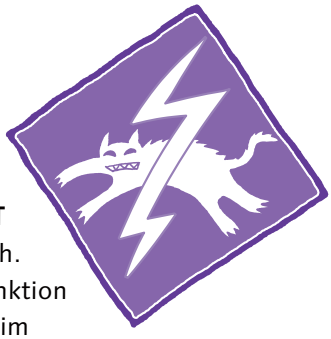
*1 Im Gegensatz zum **`print`**-Befehl musst du beim Zusammenfügen von Strings selbst an notwendige Leerzeichen zwischen den einzelnen Elementen denken.

```
eggs = 20
spam = "Jeder Tag hat *1" + str(4 + eggs) + " *1Stunden"
print(spam)*2
```

*2 **`print`** zeigt dir dann, was aus deinem **`spam`** geworden ist.

Richtig:





[Achtung]

`str(eggs)` verändert übrigens **NICHT** die Variable **`eggs`**! Ihr Inhalt bleibt gleich. Der umgewandelte Wert wird von der Funktion **`str`** zurückgegeben und an dieser Stelle im Programm verwendet.

Natürlich kannst du auch ganze Herden von **Operationen** (also Berechnungen) innerhalb solcher Funktionen wie `int()`, `float()` oder `str()` durchführen.

[Einfache Aufgabe]

Welchen Wert hat die Variable `spam` am Ende?

```
eggs = int( "2" + "4" *1)
spam = "Jeder Tag hat " + str(eggs + 12*2)*3 + " Stunden"
print( spam )
```

*1 Zuerst wird der Wert innerhalb der Klammern »berechnet«. Somit werden erst die beiden Strings zu dem String `"24"` zusammengefügt. Aus diesem String macht `int` dann die Zahl **24**, die der Variablen `eggs` zugewiesen wird.

*2 Wieder wird die Operation in der Klammer ausgeführt, bevor konvertiert wird. Die Zahl **24** aus `eggs` wird mit der Zahl **12** zusammengerechnet – zu einer gewagten **36**.

*3 Aus der berechneten Zahl **36** wird ein **String** gemacht, der zusammen mit den beiden anderen Strings zusammengefügt wird.

Das Ergebnis sieht dann so aus:

```
Jeder Tag hat 36 Stunden
```

Lustiges Konvertieren – was Python zu was macht

Es ist ganz wichtig, zumindest einmal gesehen zu haben, was Python aus unterschiedlichen Werten bei der Konvertierung macht. Schauen wir uns das mal etwas genauer mit den Funktionen **`str()`**, **`int()`** und **`float()`** an.

Die Funktion »str()« – verwandelt alles in Text

```
str()       → ''
str(0)      → '0'
str(1)      → '1'
str(12.94)  → '12.94'
str(False)  → 'False'
str(True)   → 'True'
str(None)   → 'None'
str('eins') → 'eins'
```

Hier kannst du sehen, welche Strings von der Funktion **`str()`** aus unterschiedlichen Werten »gebastelt« werden. Sogar leere oder gar keine Werte sind erlaubt und erzeugen eben einen **leeren** String.

Eine Variable ist ja auch mit einem leeren String korrekt initialisiert.
Versuchst du, eine Fließkommazahl **mit Komma** zu schreiben, führt das zu einem Fehler:

```
str(3,141)X
```

Das liegt nicht mal an der Funktion **`str()`**. Python akzeptiert eben **kein Komma** als Dezimalzeichen – nur der im Englischen übliche **Punkt** ist erlaubt.

Die Funktion »int()« – ganze Zahlen

Mithilfe von **`int()`** werden Werte in Ganzzahlen, also Integer, umgewandelt.

```
int()       → 0
int(0)      → 0
int(1)      → 1
int(12.94)  → 12
int("42")   → 42
int(False)  → 0
int(True)   → 1
```

»Leere Werte« sind auch hier erlaubt. Auch aus einem Text **`"42"`** wird korrekt eine Zahl **`42`**. Und Fließkommazahlen werden ja umgewandelt, indem der Teil nach dem Komma einfach **abgeschnitten** wird.

Bei einem **String** wie **`"42.123"`** versucht Python aber erst gar nicht, eine **`42`** zu raten – es ist wieder deine Aufgabe als Programmierer, dich darum zu kümmern. Nur aus direkt passenden Strings wie **`"42"`** erzeugt **`int()`** eine Zahl. Und auch ein Text, der eine Zahl als Wort darstellt, wie **`"one"`** oder **`"eins"`** wird von Python nicht als Zahl zurechtgeraten, sondern mit einem Fehler quittiert – genauso wie der Versuch, **`None`** in eine Zahl zu verwandeln.

```
int("42.123")X
int("one")X
int("eins")X
int(None)X
int("10vorne")X
```

Was bei **`int()`** mit dem String **`"42.123"`** nicht funktioniert, funktioniert natürlich bei **`float()`**, da die Funktion ja Zahlen mit Dezimalpunkt kennt.

Die Funktion »float()« – Fließkomma mit Punkt

Die Funktion **`float()`** ist der große Bruder von **`int()`** und für Fließkommazahlen zuständig – nur eben mit dem Punkt als Dezimalzeichen.

```
float()          → 0.0
float(0)         → 0.0
float(1)         → 1.0
float(3.141)     → 3.141
float("42")      → 42.0
float("42.123")  → 42.123
float(True)      → 1.0
float(False)     → 0.0
```

Und auch hier gibt es Werte, die nicht in eine Fließkommazahl umgewandelt werden können:

```
float("eins") X
float(None) X
float(3,141) X
```

Die Funktion »bool()« – Wahrheit oder Pflicht

Die Funktion **`bool()`** ist von solch schlichter Eleganz und Einfachheit, dass sie mit einem Absatz abgehandelt werden kann: **`bool()`** macht aus gar keinem Wert, **`None`**, einem leeren String **`''`** oder einer **`0`** (oder **`0.0`**) ein **`False`** – alles andere ist **`True`**. Selbst ein **`"False"`** oder ein Leerzeichen **`' '`** als Text wird zu einem ein **`True`**.

`Fehlt jetzt noch was? Klar!`

Du musst ja auch feststellen können, von welchem Typ ein bestimmter Wert ist. So lange du **Literale** hast, ist das natürlich banal – du siehst ja direkt, um was für einen Wert es sich handelt. Kniffliger wird es bei **Operationen** oder wenn die Daten eben nicht direkt unter deiner Kontrolle sind, wenn Daten zum Beispiel aus einer Datenbank, dem Internet oder einer Benutzereingabe kommen.

Was ist das denn für ein Typ – »type()«

Mit der Funktion **`type()`** kannst feststellen, um **welchen Datentyp** es sich bei einem Wert handelt. Dabei ist es egal, ob der Wert als Literal oder in Form einer Variablen vorliegt.

Du musst nur den Wert (bzw. die Variable) in die Klammer der Funktion schreiben. **`type()`** überprüft den Typ – todsicher und schnell.

Hier ist einmal vereinfacht das Ergebnis solcher Prüfungen dargestellt. Willst du das Ergebnis sehen bzw. ausgeben, dann gib **`type()`** am einfachsten mit einem **`print()`** aus:

```
print( type(spam) )
```

Und praktisch ausprobiert sieht das so aus:

Code	Ausgabe
`type(42)`	`<class 'int'>`
`type(42.123)`	`<class 'float'>`
`type("Hallo")`	`<class 'str'>`
`type(False)`	`<class 'bool'>`
`type( 10 > 1`*1 `)`	`<class 'bool'>`
`type()`*2	FEHLER!

*1 Innerhalb der Klammer einer Funktion – also auch bei **`type()`** – kannst du Operationen bzw. Berechnungen verwenden. Es wird das **Ergebnis** der gesamten Operation untersucht: Da hier 10 unzweifelhaft größer als 1 ist, ergibt dieser Vergleich im Ergebnis wahr, also **`True`**. Und **`type`** stellt somit ganz richtig fest, dass wir einen **booleschen Wert** haben.

*2 Die Funktion **`type()`** verträgt es nicht, wenn sie leer, also **ohne Inhalt**, aufgerufen wird. Dann kommt es zu einem Fehler.

Die Ausgabe in der Form **`<class 'IRGENDWAS'>`** erscheint auf den ersten Blick vielleicht etwas seltsam. Du musst dazu wissen, dass so ziemlich alles in Python als ein **Objekt** angesehen wird. Daher rührt die Angabe **`class`**. Bei diesen Klassen, **`class`**, handelt es sich um so etwas wie **Vorlagen**, was aus der objektorientierten Programmierung kommt. Hinter jedem Datentyp steht also eine entsprechende Klasse. Und genau diese Klasse beschreibt, was die jeweiligen Datentypen ausmacht.

Ui, das sieht aber kompliziert aus!

Keine Sorge: Für einen **Vergleich** in einem Programm, also um zu testen, ob ein bestimmter Typ vorliegt, ist der Aufruf viel leichter. So prüfst du beispielsweise, ob der Wert einer Variablen vom Typ her ein String ist:

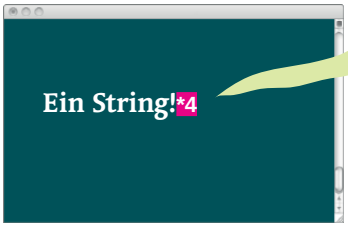
*1 Das hier ist ein `if`. Damit kannst du Entscheidungen treffen, die von einem **Vergleich** abhängen. Das `if` wirst du später noch genauer kennenlernen.

*2 Hier fragst du den **Typ** ab, den der Wert der Variablen gerade hat, …

```
eingabe = "Hallo"
if*1 type(eingabe)*2 == str*3:
    print("Ein String!")*4
```

*3 … und vergleichst ihn mit dem **Typ String** – der ganz einfach als `str` (ohne Anführungszeichen) geschrieben wird.

*4 Siehe da: Es ist ein String!



Auch auf andere Typen kannst du so testen. `bool` steht für einen Wahrheitswert, `int` für einen Integer und `float` für eine Fließkommazahl.
Alle Typen werden **ohne Anführungszeichen** geschrieben.

Aber im Moment genügt, dass du feststellen kannst, um welchen Datentyp es sich handelt.

Probier's doch mal aus

[Einfache Aufgabe]
Finde heraus, um welchen Datentyp es sich jeweils handelt.

```
spam = "10" * 42    *1
spam = int("42") > 30    *2
egg = 12.0 * 3    *3
ergebnis = spam * egg    *4
```

*1 Das Ergebnis ist ein ziemlich langer **String** in der Art "10101010101..." (wobei wir uns den Rest des Strings sparen).

*2 Klar, das Ergebnis aus einer **Vergleichsoperation** zweier Zahlen mit `>` ist (in diesem Fall) wahr, also `True`, und damit ein boolescher Wert, also **bool**.

*3 Einmal `float` und einmal `int`? Klar, das Ergebnis ist in dem Fall praktisch immer `float`, hier `36.0`

*4 In einer Berechnung mit einer Zahl wird unser boolescher Wert in `spam` zu `0` oder `1`. Unser `True` wird also zu `1` bzw. in der Operation zu `1.0` und das Ergebnis ist `36.0` – also wieder `float`.

Mit `print(type(name_der_variable))` kannst du dir den Typ jeweils ausgeben lassen.

Und was ist noch wichtig?

Eine Variable selbst ist nicht dauerhaft auf eine bestimmte Art von Wert bzw. einen Datentyp festgelegt. Im Gegenteil darf jede Variable lustig ein »Bäumchen wechsel dich« spielen.

```
spam = 42
spam = "Schrödinger"
spam = True
```

Auch wenn Variablen unterschiedliche Datentypen haben und **wiederverwendet** werden können, gilt folgender Hinweis:

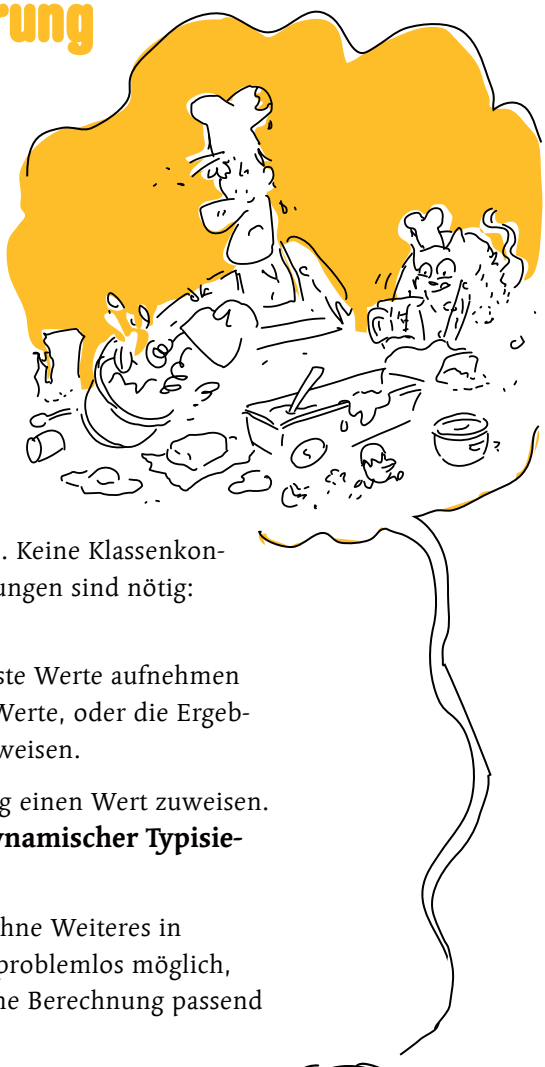
[Achtung]

Es ist **keine gute Idee,** Variablen »recyceln« zu wollen. Nimm lieber eine **neue Variable**.

Python kümmert sich um Variablen, die nicht mehr benötigt werden. Es wird also kein Speicher durch alte Variablen belegt. Die Anzahl der Fehler, die durch recycelte Variablen entstanden sind, ist hingegen legendär. Viele Entwickler können davon ein leidvolles Lied singen, und es gibt kaum bessere Möglichkeiten, sich bei Kollegen unbeliebt zu machen.

Schrödinger, das Zen of Python sagt dir: »Readability counts. – Die Lesbarkeit zählt.«

Syntax, Variablen, Datentypen und dynamische Typisierung



Lass uns kurz rekapitulieren …

- Python ist nach einer (nein, **der**) britischen **Komikertruppe Monty Python** benannt. Wichtige Empfehlungen und Regeln sind in den sogenannten PEPs festgelegt.
- Die **Syntax** ist »leichtgewichtig«. Gerade auf die schlecht zu tippenden geschweiften Klammern oder das »schicke« Semikolon wurde weitgehend verzichtet. Jeder Befehl steht in einer eigenen Zeile.
- Ein Python-Programm schreibst du einfach so. Keine Klassenkonstrukte, keine Importe oder andere Vorbereitungen sind nötig: einfach schreiben, speichern, ausführen.
- **Variablen** sind Speicher, die unterschiedlichste Werte aufnehmen können. Du kannst ihnen Literale, also feste Werte, oder die Ergebnisse von Berechnungen und Operationen zuweisen.
- Jeder Variablen musst du vor der Verwendung einen Wert zuweisen. Den Datentyp erkennt Python selbst, dank **dynamischer Typisierung**.
- Unterschiedliche **Datentypen** dürfen nicht ohne Weiteres in Berechnungen vermischt werden. Es ist aber problemlos möglich, den Typ eines Wertes festzustellen und für eine Berechnung passend zu machen.
- Mit `print` kannst du Werte komfortabel ausgeben, getrennt durch ein Komma verstehen sich hier sogar unterschiedliche Datentypen wunderbar.

Zeit für eine Pause …?
… keine Pause!

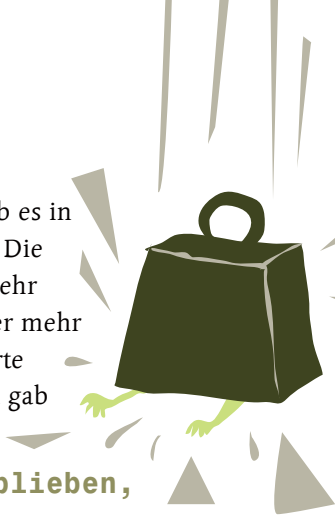
—ACHT—

Objektorientierte Programmierung

Von Klassen, Objekten und alten Griechen

Python ist nicht nur klasse, Python kann auch wunderbar mit Klassen und Objekten umgehen! Du lernst in diesem Kapitel nicht nur, was es damit auf sich hat und wie du schneller und noch fehlerfreier programmieren kannst. Du erfährst auch noch, was die alten Griechen damit zu tun haben. Und dabei ist OOP nicht mal halb so schwer, wie es die ganzen Fachbegriffe vermuten lassen.

Die gute, alte Softwarekrise



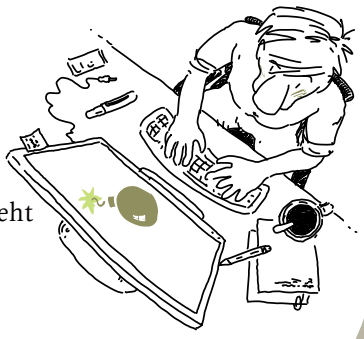
Es ist noch gar nicht sooo lange her, da gab es in der Entwicklung von Programmen eine sogenannte **Softwarekrise**: Die Programme wurden immer umfangreicher und mächtiger. Immer mehr Entwickler arbeiteten gleichzeitig an einem Programm. Es gab immer mehr Änderungswünsche noch während der Entwicklungsphase. Das führte dazu, dass es immer mehr Fehler und Probleme in den Programmen gab und Projekte dadurch immer häufiger aus dem Ruder liefen.

Und irgendwie ist das bis zum heutigen Tag so geblieben, zumindest ungefähr.

In Python ausgedrückt:

```
software_krise = umfang_code * anzahl_entwickler * aenderungen
```

Vielleicht kennst du das ja schon selbst: Die erste Idee zu einem neuen Programm ist schnell umgesetzt – mit **wenig Code und sehr stabil**. Aber je ausgefeilter das Programm wird, desto **mehr Sonderfälle** und Eventualitäten kommen dazu, desto **mehr Code** entsteht dadurch und desto **mehr Fehler** schleichen sich ein.



Nö. Fehler?

Bei mir?

Noch nie gesehen ...

Retter gesucht? Retter gefunden: OOP!

Eine neue Art zu programmieren, eine neue **Technik** bzw. Herangehensweise musste her: die **o**bjekt**o**rientierte **P**rogrammierung, kurz **OOP**!

Und was ist das?

In der OOP unterteilst du deine Programme in **kleinere Einheiten** – das sind in Python und den meisten objektorientierten Sprachen **Klassen** und **Objekte** (die aus genau diesen Klassen erzeugt werden). Diese kleineren Einheiten sind ...

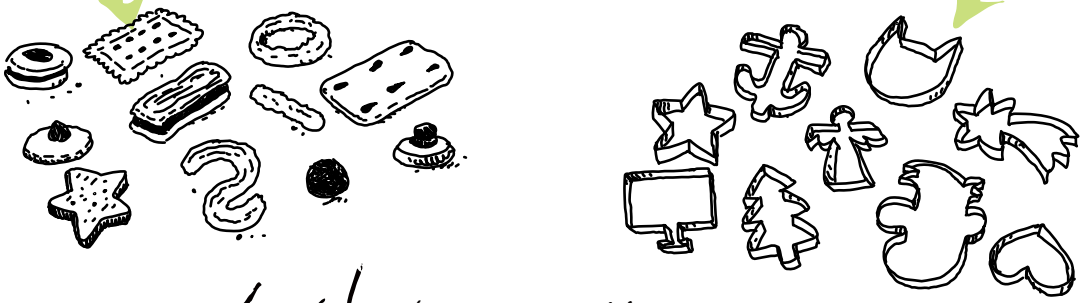
- … naturgemäß leichter zu überblicken als ein riesiges, zusammenhängendes Programm mit Hunderten (oder Tausenden) von Zeilen, das nur von einigen Funktionen unterbrochen wird.
- … unabhängig voneinander und vom restlichen Programm. Du selbst kannst festlegen, auf welche Teile von außen (also dem Rest des Programms und auch von **anderen Klassen**) zugegriffen werden darf.
- … dadurch auch wesentlich robuster gegenüber Änderungen, die irgendwo im Programm gemacht werden.

Das klingt ein bisschen nach Funktionen?

Funktionen sind gar nicht so weit entfernt von **Klassen** und **Objekten**: Sie stehen für sich selbst und existieren getrennt von anderen Elementen in einem eigenen Bereich. Eine Funktion ist zunächst eine abstrakte Sache: Nur durch die **Definition** einer Funktion tut sich erst mal noch gar nichts. Erst durch den **Funktionsaufruf** wird eine Funktion mit Leben gefüllt bzw. tritt in Erscheinung.

> Mit Klassen und Objekten ist das durchaus ähnlich. Eine Klasse selbst ist zunächst einmal nur die **Definition** – eine Vorlage oder ein **Bauplan**: etwas Programmcode, einige Variablen und das ist es auch schon. Erst wenn aus einer Klasse **ein Objekt** erstellt wird (man sagt auch **instanziiert** wird), kommt Leben in die Sache – so wie beim **Funktionsaufruf** einer Funktion.

Stell dir das so vor wie beim Plätzchenbacken: Du hast die ganzen Ausstechformen, die festlegen, wie so ein Plätzchen grundsätzlich aussehen kann – das ist die Klasse. Und mithilfe dieser Ausstechformen (aka Klassen) machst du dann konkrete Plätzchen, die jeweils die gleiche Form haben – das sind die Objekte. Und nur die Plätzchen (aka die Objekte) werden tatsächlich gegessen bzw. verwendet.



Lecker! Aber wo sind jetzt die Vorteile von diesen Klassen und Objekten gegenüber Funktionen?

Eine Funktion wird aufgerufen, arbeitet ihren Code ab und liefert vielleicht Werte zurück. Danach **verschwindet** alles im Daten-Nirwana. Ein **Objekt** hingegen (mit allem, was dazugehört) »lebt« so lange, wie das Programm läuft, und behält alle seine Werte während seiner »Lebenszeit«.

[Zettel]
Objekte leben länger, und zwar im Gegensatz zu kurzlebigen Funktionen so lange, wie das Programm läuft!

Eine Funktion ist **eine** Funktion – fertig.

Ja, klingt ja auch logisch ...

Eine **Klasse**, aus der dann die **konkreten Objekte** erzeugt werden, kann hingegen nahezu **beliebig viele,** unterschiedliche, sogenannte **Methoden** haben. So eine Methode selbst ist einer Funktion ziemlich ähnlich, nur gehört sie eben zu einer Klasse.

[Zettel]
Eine Klasse (und damit ein Objekt) kann beliebig viele **Methoden** haben.

Eine Funktion kann eigene Variablen haben. Auch können einer Funktion Werte übergeben werden und sie kann Werte zurückliefern. Nach dem Aufruf der Funktion »verschwinden« diese Variablen aber – samt ihren Werten.

Klassen können dagegen ganz **besondere Variablen** haben: sogenannte **Attribute**. Das sind Variablen, auf die sogar von außen zugegriffen werden kann und die **so lange existieren**, wie das Programm läuft, und die nicht schon nach einem Aufruf wieder Geschichte sind und alle Werte verlieren.

[Zettel]
Ein Objekt kann **Attribute** haben, die über den Aufruf hinaus verfügbar sind und ihre **Werte behalten** – zumindest bis zum Programmende.

Der objektorientierten Programmierung liegt ein ganz bestimmtes **Konzept** zugrunde: **Klassen** und **Objekte**. Der Gedanke dabei ist ganz einfach und stammt von den alten griechischen Philosophen.

[Hintergrundinfo]

Alles lässt sich in **Klassen** einteilen. Sie beschreiben, wie etwas **aussieht** und wie es sich **verhalten** kann. Eine Klasse ist eine Art Bauplan oder eine Schablone, mit der alle **Eigenschaften** und **Fähigkeiten** festgelegt werden. Klassen sind abstrakt – eben nur der Plan. Aus diesem Bauplan muss erst ein **konkretes Etwas**, ein **Objekt**, erzeugt werden.

Ganz konkret – die Sache mit Klassen und Objekten

Wie sieht denn jetzt so eine Klasse mit so einem Objekt aus?

Das hier ist eine grundlegende **Klasse**:

```
class Spam:
    pass
```

Eine wirklich ganz **einfache** Klasse. Sie hat keine Fähigkeiten, keine Merkmale und auch keine Attribute. Und von einem zugehörigen Objekt ist hier erst mal noch nichts zu sehen.

Diese Klasse kann noch nichts machen, ist aber immerhin schon mal eine echte Klasse. Es ist übrigens **üblich**, den Namen einer Klasse in **CamelCase** großzuschreiben.

Geben wir unserer Klasse noch eine einfache **Methode**, die etwas mit `print` ausgibt:

```
class Spam:
    def eine_methode(self):
        print(42)
```

*1 Deine erste Methode. Und ja, eigentlich ist sie wie eine Funktion aufgebaut.

*2 Das ist etwas Neues: ein besonderer Parameter, den **jede** Methode einer Klasse benötigt.

Das `pass` brauchen wir jetzt natürlich nicht mehr.

Was hat es denn mit diesem `self` *auf sich?*

Eine **Methode** ist einer Funktion ja recht ähnlich, bis auf die Tatsache, dass sie eben zu einer Klasse (bzw. einem konkreten Objekt) gehört. Bei jedem Aufruf einer solchen Methode werden **intern** Informationen über das Objekt an die Methode übergeben. Und damit das auch klappt, muss der Parameter `self` angegeben werden.

[Hintergrundinfo]
Der Name `self` als interner Parameter ist übrigens nicht fest vorgegeben. Es ist lediglich eine **Konvention**. Oftmals wird anstelle von `self` auch `this` geschrieben. Du könntest aber auch beliebige Namen verwenden.

Von der ersten Klasse zum ersten Objekt

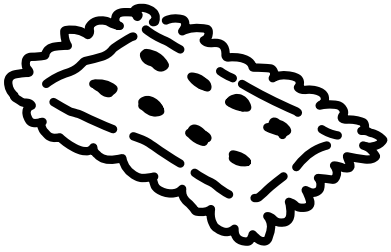
Bis jetzt hast du nur die (doch eher abstrakte) **Klasse** erstellt. Und selbst wenn du alles speicherst und als Programm ausführst, passiert erst mal gar nichts – genauso wie bei einer Funktion.

Du musst jetzt aus der Klasse – quasi als Vorlage – ein konkretes **Objekt erschaffen,** das nennt man **instanziieren**. Dazu weist du die Klasse einer Variablen zu. Das geschieht natürlich außerhalb der Klasse und – genauso wie bei einer Funktion – **nach** der Definition der Klasse.

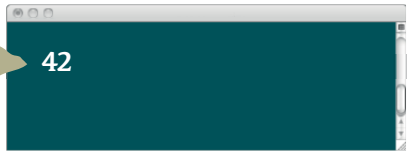
```
eggs = Spam()
```

Eine klassische Zuweisung. Nur wird hier der Variablen kein einfacher Wert, sondern die **Klasse `Spam`** zugwiesen. Und damit entsteht ein neues **Objekt**.

Jetzt kannst du mit diesem Objekt `eggs` arbeiten und **alle Fähigkeiten der Klasse** nutzen. `Wichtig:` Für den Zugriff von außen im Programm musst du immer den **Namen des Objekts** und den **Namen des gewünschten Elements** schreiben, getrennt durch einen **Punkt** – und bei einer Methode natürlich mit runden Klammern. So sieht das dann bei unserem Objekt `eggs` für den Aufruf der Methode `eine_methode` aus:



```
eggs.eine_methode()
```



[Achtung]
Ähnlich wie bei einer Funktion muss die Klasse bereits definiert sein, **bevor** daraus ein Objekt instanziiert werden kann

Alles auf Anfang – die Methode »__init__«

Das ist aber natürlich noch nicht alles! Klassen können **besondere Methoden** haben, die schon fast magische Fähigkeiten haben. Eine dieser besonderen Methoden ist `__init__`, die immer dann aufgerufen wird, wenn ein neues Objekt aus einer Klasse erstellt wird:

```
class Spam:
    def __init__(self):
        print("Hallo in der Welt der OOP!")
    def eine_methode(self):
        print(42)
```

*1 Wie der Name `__init__` schon ein wenig vermuten lässt, ist das hier eine besondere Methode: Sie wird **einmalig** aufgerufen, wenn ein **neues Objekt** aus einer Klasse erstellt wird.

*2 Und auch diese besondere Methode benötigt zwingend den Parameter `self`.

```
eggs = Spam()
```

*1 Bereits hier, bei der **Erschaffung** des Objekts, tut sich schon etwas – die Methode `__init__` wird abgearbeitet!

```
Hallo in der Welt der OOP!*1
```

```
eggs.eine_methode()
```

```
42
```

Jedes Mal, wenn aus einer Klasse ein **neues Objekt** erstellt wird, wird **einmalig** die Methode `__init__` aufgerufen, und zwar automatisch und als **Allererstes**!

[Hintergrundinfo]

In der Methode `__init__` kann alles gemacht werden, was (für ein Objekt) in einem Programm **beim Start** von Bedeutung sein könnte, zum Beispiel die Initialisierung von Attributen, die Eingabe wichtiger Parameter oder auch der Aufbau einer Datenbankverbindung. Dabei müssen all diese Funktionalitäten gar nicht in `__init__` selbst sitzen. Es macht mehr Sinn, das in eigene Methoden zu schreiben und diese dann von `__init__` aufrufen zu lassen. In anderen Programmiersprachen kennt man so etwas als **Konstruktor**.

Cool!

Dein erstes Attribut

Jetzt definieren wir zum ersten Mal auch noch ein echtes **Attribut**, und zwar beim Erstellen eines Objekts in der Methode `__init__`, das ist schließlich ein idealer Ort für so etwas. In unserer Methode `eine_methode` greifen wir dann auf dieses Attribut zu:

```
class Spam:
    '''Eine Klasse, nur als einfaches Beispiel'''*1
    def __init__(self):
        self.ein_attribut = 42*2
        print("Hallo in der Welt der OOP!")
    def eine_methode(self):
        print(self.ein_attribut*3)
```

*1 Hier haben wir jetzt auch noch einen Docstring hinzugefügt. Auch das kennst du ja von Funktionen.

*2 Das ist ein **Attribut**: also eine Variable, die **zum Objekt** gehört, und eben nicht nur Teil einer Methode ist. Das erkennst du an dem `self` vor dem Namen. Wir weisen unserem Attribut einen beliebigen Wert zu. Wir nehmen mal die `42`.

*3 Hier greifst du in **einer anderen Methode** auf das Attribut zu. Auch hier mit der Angabe von `self` als `self.ein_attribut`.

[Zettel]
Willst du auf ein Element **des eigenen Objekts** zugreifen, brauchst du den Bezug dazu in Form der Angabe `self`. Also `self.ein_attribut` oder `self.eine_methode()`.

Genauso wie auf eine Methode kannst du **von außen** auf das **Attribut zugreifen**! Das funktioniert, weil so ein Attribut nach seiner Initialisierung eben nicht nur beim Aufruf einer Methode existiert, sondern solange das Objekt existiert.

Das kann dann so aussehen:

```
eggs = Spam()
eggs.eine_methode()
print(eggs.ein_attribut*1)
```

*1 Hier greifst du direkt auf das **Attribut** zu.

```
Hallo in der Welt der OOP!
42
42*1
```

Es ist dir bestimmt schon aufgefallen:

Innerhalb der Klasse greifst du auf (eigene) Elemente mit `self.attribut` bzw. `self.methode()` zu. **Außerhalb** machst du das mit dem Namen des Objekts und dem Namen des Attributs bzw. der Methode: `name_des_objekts.attribut` bzw. `name_des_objekts.methode()`.

Übrigens: Jedes Objekt, das du erschaffst (also **instanziierst**), besitzt **alle** Attribute, Werte und Methoden, die auch die zugrunde liegende Klasse hat. Die Klasse ist tatsächlich der exakte Bauplan, nach dem das Objekt dann genauso erschaffen wird!

Es gibt auch ganz schnöde Variablen

Innerhalb von Methoden kann es auch ganz normale, **kurzlebige** Variablen geben, die nach dem Aufruf der Methode im Daten-Nirwana verschwinden – ganz so, wie das auch bei Funktionen der Fall ist.

Ohne `self` vor einer Variablen geht Python davon aus, dass es sich um eine ganz normale Variable der Methode handelt:

*1 Es muss ja nicht immer **Spam** sein!

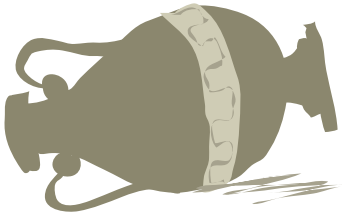
```
class Eggs:*1
    def __init__(self):
        self.ein_attribut = 42
    def eine_methode(self):
        verdoppelt*2 = self.ein_attribut * 2
        print(self.ein_attribut)
        print(verdoppelt)*3
mein_objekt = Eggs()
mein_objekt.eine_methode()
print(mein_objekt.verdoppelt)*4X
```

*2 Das hier ist kein **Attribut**, sondern eine ganz **normale Variable**, gut zu erkennen an dem fehlenden `self` vor dem Namen.

*3 Nur **innerhalb** dieser Methode kannst du mit dieser (normalen) Variablen, `verdoppelt`, arbeiten.

*4 Das **geht gar nicht**! Es gibt nämlich kein Attribut `mein_objekt.verdoppelt`!

[Achtung]
Auf die Variable `verdoppelt` könntest du **von außen nicht** zugreifen, da sie ja nur im kurzen Moment des Aufrufes der Methode existiert und danach im Daten-Nirwana verschwindet, so, wie du das von den Variablen in Funktionen her kennst.



Mehr Infos dank Docstring

Du kannst dir auch den **Docstring** deines Objekts ausgeben lassen und sogar die **Hilfe** mit `help()` aufrufen, die dir noch mehr Informationen ausgibt. Dabei spielt es keine Rolle, ob du dazu das Objekt **oder** seine Klasse angibst:

```
print(eggs.__doc__) # print(Spam.__doc__)
```

```
Eine Klasse, nur als einfaches Beispiel
```

```
help(eggs) # genauso geht es mit der Klasse: help(Spam)
```

```
Help on spam in module __main__ object:
class Spam(builtins.object)
 |  Eine Klasse, nur als einfaches Beispiel
 |  Methods defined here:
 |  __init__(self)
 |    Initialize self. See help(type(self)) for accurate signature.
 |  eine_methode(self) |  eine_methode(self)
```

Durch den Aufruf von `help` werden dir der **Docstring** und weitere Informationen, wie zum Beispiel alle Methoden, ausgegeben. Aus Platzgründen ist die Ausgabe von `help` hier etwas gekürzt.

Aber weiter, Schrödinger:

[Zettel]

Denk daran, du arbeitest hier nicht mit der Klasse. Sie wird nur als eine Art **Vorlage** verwendet, mit der ein (konkretes) Objekt erschaffen bzw. **instanziiert** wird.

Und noch etwas, du bist auch **nicht** darauf **beschränkt**, nur ein Objekt einer Klasse zu haben. Du kannst dir **beliebig viele** Objekte aus einer Klasse erschaffen:

```
noch_ein_objekt = Spam()
leckeres_essen = Spam()
```


Jedes Objekt der gleichen Klasse hat die **gleichen** Attribute und Methoden. Damit sind auch alle einprogrammierten **Werte** für alle Objekte einer Klasse erst mal gleich. **Nach** der Instanziierung (also der Erschaffung) sind diese Objekte aber vollkommen **unabhängig** voneinander, und Änderungen an einem Objekt wirken sich nicht auf die anderen Objekte aus.

Das machen wir mal ganz schnell:

```
noch_ein_objekt.ein_attribut = 13
leckeres_essen.ein_attribut = 7
```

Hier wurde bei jedem Objekt das Attribut **`ein_attribut`** geändert.

Der Aufruf der Methoden erfolgt ja genauso mit dem jeweiligen Objektnamen und dem jeweiligen Element bzw. der jeweiligen Methode. Natürlich könntest du auch das Attribut wieder direkt abfragen.

```
noch_ein_objekt.eine_methode()
leckeres_essen.eine_methode()
```

```
13
7
```

Du siehst, beide Objekte geben über ihre Methode **`eine_methode`** unterschiedliche Werte aus – kein Wunder, denn ein paar Zeilen weiter oben haben wir ja den **Attributen** jeweils unterschiedliche Werte zugewiesen!

Das Orakel von Delphi

Es wird Zeit für dich, deine **erste Klasse** zu programmieren. Und da das **Konzept** der Klassen und Objekte von den alten Griechen kommt, was liegt da näher, als etwas klassisch Griechisches zu programmieren:

Das Orakel von Delphi!

Das Orakel von Delphi machte in der Antike Weissagungen, entweder verklausuliert, in etwas kryptischer Art oder einfach als »ja« oder »nein«. Von überall her kamen Menschen zu dem Tempel des Gottes Apoll, wo die Weissagungen gemacht wurden.

[Einfache Aufgabe]
Deine Aufgabe ist es, ein Orakel zu schreiben, das als Weissagung `"Ja"`, `"Nein"` oder `"Vielleicht"` spricht bzw. als Wert zurückgibt, …

… und zwar so richtig als Klasse mit **Attributen**, **Methoden** und dann natürlich mit einem **Objekt**.

Es gibt aber noch eine **Besonderheit** bei der **Aufgabe** zu berücksichtigen:

[Schwierige Aufgabe]
Es darf niemals die gleiche Antwort zweimal hintereinander gegeben werden!

Was brauchst du dazu:

- erst einmal eine **Klasse**
- dann ein **Attribut**, in dem du als Liste oder als Tupel die möglichen Antworten speicherst
- und noch ein **weiteres Attribut**, in dem du die letzte gegebene Antwort speicherst, denn du musst ja verhindern, dass die Antwort sofort wiederholt wird
- und dann natürlich eine **Methode**, in der eine neue Antwort gesucht wird, die nicht direkt zuvor schon gegeben wurde

Schrödinger, übernehmen Sie!

Noch ein Tipp: Mit **`random.choice(eine_liste)`** kannst du dir ein zufälliges Element aus einer Liste oder einem Tupel zurückgeben lassen.

Übrigens, wundere dich nicht: Auch ein Objekt kann sich Daten nur merken, solange das Programm läuft. Beim nächsten Programmstart ist alles wieder auf Anfang gesetzt.

Also, dann wollen wir mal ...

Für den Zufall bei der Weissagung wird das Modul **`random`** benötigt, also wird es importiert.

```
import random
```

Dann folgt die Klasse, wir nennen sie mal **`Orakel`**:

```
class Orakel:
```

Als Nächstes kommen die **beiden Variablen**, genauer gesagt die beiden **Attribute**, die in **`__init__`** angelegt werden:

```
    def __init__(self):
        self.antworten = 'Ja', 'Nein', 'Vielleicht'
        self.alte_antwort = None
```

Das erste Attribut ist ein Tupel mit den **möglichen Antworten**, in dem zweiten Attribut wird die **alte Antwort** gespeichert, jetzt ist es erst noch leer. Anstelle von **`None`** würde auch **`''`** als leerer String gehen.

Danach folgt die Methode für die Weissagung, natürlich mit dem Parameter **`self`**:

```
def weissagt(self):
```

Hier kommt die Logik, mit der ein zufälliger Wert aus dem Tupel **`self.antworten`** geholt wird. Aber erst wenn dieser Wert ungleich der vorherigen Antwort ist, wird die **`while`**-Schleife verlassen und das gültige Ergebnis zurückgegeben:

```
while True:
  auswahl = random.choice(self.antworten)
  if auswahl != self.alte_antwort:
    self.alte_antwort = auswahl
    break
return auswahl
```

Erst wenn die Auswahl **ungleich** der alten Antwort ist, wird die Schleife verlassen. Zuvor wird noch schnell die aktuell gewählte Antwort in dem Attribut **`alte_antwort`** gespeichert.

Das war jetzt schon etwas tricky!

Die **`while`**-Schleife wird erst wieder verlassen, wenn die alte Antwort **ungleich** der neuen Antwort bzw. Auswahl ist. Sind beide hingegen noch **gleich**, dann wird die **`while`**-Schleife **gleich noch mal** durchlaufen. Und zwar so lange, bis ein neuer Wert gefunden wurde, der in der Runde davor noch nicht verwendet wurde!

Genau!

Zum Schluss wird das neue Ergebnis, der Spruch des Orakels, mit **`return`** zurückgegeben – **`fertig!`**

Die erste Klasse am Stück – gleich mal etwas reloaded

Hier noch einmal alles am Stück, gleich mit ein paar kleinen **Verbesserungen**, nachdem du jetzt sicher schon etwas sattelfester bist:

```
import random
class Orakel:
    '''Das Orakel gibt dir Antworten auf deine Fragen'''*1
    def __init__(self):
        self.antworten = 'Ja', 'Nein', 'Vielleicht'
        self.alte_antwort = None
    def weissagt(self):
        while True:
          auswahl = random.choice(self.antworten)
          if auswahl != self.alte_antwort:
            self.alte_antwort = auswahl
            return auswahl*2
```

*1 Ein Docstring ist besser als jede Dokumentation, die ja doch nie geschrieben wird. Und wenn es komplizierter wird, kannst (und solltest) du hier mehr zu deiner Klasse schreiben.

*2 Anstelle des **`break`** kannst du auch gleich **`return auswahl`** schreiben und auf diesem Weg die Schleife und auch gleich die Methode verlassen.

Es ist übrigens immer besser, eine Klasse möglichst **allgemein** zu halten (genauso wie eine gute Funktion). Eine Klasse sollte so geschrieben werden, dass sie nur die Aufgaben übernimmt, die in der Klasse auch wirklich **Sinn machen**.

Stell dir mal vor, du hättest die **Ausgabe** bereits in der Klasse gemacht! Was wäre dann, wenn der Orakelspruch **als Teil** eines anderen Textes ausgegeben werden sollte oder wenn mehrere Ergebnisse in eine Zeile geschrieben werden sollen? Programmierst du die **Ausgabe** bereits **in der Klasse**, müsstest du jedes Mal deine Klasse umschreiben – und dann würde sie möglicherweise für die bisherigen Aufrufe **nicht mehr passen**.

> Ein wichtiger und zentraler Punkt der OOP ist die **Wiederverwendbarkeit** von Code. Und da ist eine Klasse wesentlich besser, wenn sie sich auf das Wesentliche beschränkt und **nur das macht**, was von ihr eigentlich zu erwarten ist.

In unserer Klasse ist das eine Antwort in Form eines Rückgabewertes – und eben nicht (unbedingt) eine feste, formatierte Ausgabe mit **`print`**! **Zumindest nicht als einzige Variante!** Denn eine Klasse kann ja sehr wohl verschiedene Methoden haben.

Also weiter, deine nächste Aufgabe wartet!

Das erste eigene Objekt

Du musst noch aus der Klasse ein **Objekt** erschaffen und eine Ausgabe machen. Am besten lässt du gleich mehrmals das Orakel sprechen, um zu sehen, ob auch alles so funktioniert, wie erwartet.

[Einfache Aufgabe]
Erschaffe, also **instanziiere**, ein Objekt aus der Klasse **`Orakel`** und lass dir zehn Weissagungen machen, die in einer Zeile ausgegeben werden sollen.

*1 Das Objekt heißt **`mein_orakel`**.

*2 Die Klasse wird einfach der Variablen zugewiesen, so entsteht ein Objekt daraus.

```
mein_orakel*1 = Orakel()*2
for i in range(10):*3
    print(mein_orakel.weissagt()*4, end = " "*5)
```

*3 Hier beginnt eine Schleife, …

*4 … ruft zehnmal die Methode auf und gibt den Rückgabewert, die Antwort des Orakels, aus.

*5 So wird alles in eine Zeile geschrieben – was natürlich jedes Mal anders aussieht, die Antworten werden ja zufällig ausgewählt.

```
Vielleicht Ja Nein Vielleicht Nein Vielleicht Ja Nein Vielleicht Ja
```

Aber mal ehrlich, ich fände es schon ganz praktisch, die Weissagungen auch gleich ausgeben zu können.

`Das ist kein Problem.`
Du kannst einfach eine **weitere Methode** schreiben, die auch eine Ausgabe macht.

Echt jetzt? Das geht?

Klar, du schreibst einfach **eine weitere Methode**, die eben eine schön formatierte Ausgabe mit `print` macht! Schließlich kannst du nicht nur von außen auf die Methoden der Klasse zugreifen. Du kannst weitere Methoden schreiben, die auf die bestehenden Methoden und Attribute **zugreifen** können. Du kannst also eine Methode für die Ausgabe schreiben, die sich eine Weissagung von der Methode `weissagt` erstellen lässt!

Cooool!

[Schwierige Aufgabe]
Schreib eine Methode `ausgabe`, die eine Weissagung ansprechend formatiert mit `print` ausgibt.

[Notiz]
Es spielt übrigens **keine Rolle**, in welcher Reihenfolge die Methoden und Attribute innerhalb der Klasse geschrieben sind. **Üblich** ist es, die Attribute an den Anfang zu stellen und die Methoden darunter. Die Methode `__init__` ist dabei sinnvollerweise die erste Methode.

Dann mach ich das mal!

Also erst die **Definition**, als Parameter muss wieder `self` angegeben werden:

```
def ausgabe(self):
```

Dann eine schicke Linie:

```
    print('+' * 30)
```

Und hier kommt die Ausgabe der Weissagung. Mit `f` vor dem Text können Variablen und Funktionsaufrufe mit geschweiften Klammern direkt in den Text geschrieben werden. Das funktioniert natürlich genauso mit Methoden bzw. deren Rückgabewerten.

```
print(f"Das Orakel spricht '{self.weissagt()}'")
```

Über den angegebenen Parameter **`self`** greifst du auf eigene Methoden der Klasse zu. Da eine Methode wie eine Funktion ist, darfst du die runden Klammern nicht vergessen.

Dann noch eine weitere Linie:

```
print('+' * 30)
```

Übrigens, Schrödinger, du könntest natürlich auch zuerst die Zeile mit der Weissagung erzeugen, in einer Variablen speichern und dann daraus die Länge der Zeilen dynamisch generieren.

Coole Idee!
So in etwa?

```
def ausgabe(this):
    text = f"Das Orakel spricht '{this.weissagt()}'"
    print('+' * len(text))
    print(text)
    print('+' * len(text))
```

Mit dem Aufruf sieht es dann fertig so aus:

```
mein_orakel = Orakel()
mein_orakel.ausgabe()
```

[Notiz]
Die bisherige Methode **`weissagt`** funktioniert nach wie vor und kann wie bisher aufgerufen werden. Die **Erweiterung** in Form einer zusätzlichen Methode hatte hier **keinen Einfluss** auf die bestehenden Elemente der Klasse!

Eine der größten Gefahren (unglaublich, aber wahr) in der Programmierung ist die **Veränderung** von bestehendem Code. Dabei kann es genügen, ein paar neue Zeilen in den Programmfluss einzufügen – aus Versehen wird eine bestehende Variable beeinflusst oder Bedingungen werden verändert, und schon kommt es zu unvorhergesehenen Fehlern. **Willkommen in der Softwarekrise …**

Bei Klassen hingegen ist es fast **gefahrlos**, neue Methoden **hinzuzufügen**. Denn die anderen Methoden werden dadurch ja nicht verändert. Klassen mit ihren unterschiedlichen Methoden sind hier klar im Vorteil! **Mehr Code – weniger Fehler.**

Vorsichtig musst du natürlich sein, wenn du in neuen Methoden **die Werte** bestehender Attribute änderst. Das kann dann natürlich Einfluss auf das bestehende Programm haben!

[Notiz]

Den Parameter `self` brauchst du zwingend in jeder Methode. Wie du ihn tatsächlich benennst, ist dagegen vollkommen dir überlassen. Du **könntest** (nein, bitte tue es nicht!) ihn sogar in jeder Methode unterschiedlich benennen: in einer Methode zum Beispiel `this`, in einer anderen Methode eben `self`. Natürlich solltest du das **einheitlich** machen, und `self` ist eine gute Wahl, aber so musst du dich zumindest nicht wundern, woher plötzlich »irgendwelche ganz anderen« Parameter in fremden Klassen kommen. Das ist meist `self` – nur anders benannt.

Eine Frage habe ich aber noch!

Warum kann ich in meiner Klasse nicht anstelle von `self` (oder `this`) einfach den Namen des Objekts, zum Beispiel `mein_orakel` verwenden?

Das macht wenig Sinn. Also rein von der Syntax her würde das sogar gehen, denn es wäre dann nur ein anderer Name als `self` und hätte trotz des gleichen Namens keinen weiteren Bezug zu deinem späteren Objekt. Sinnvoll wäre das also nicht – außerdem weißt du ja beim Schreiben einer Klasse sowieso noch gar nicht, wie ein entsprechendes Objekt später heißt. Und da du mehrere Objekte haben kannst, macht so etwas noch weniger Sinn. Die Namensgleichheit hätte **keinerlei Vorteil** gegenüber einer klassischen, abstrakten Bezeichnung wie `self` oder `this`.

Orakel reloaded – das Attribut ändern

Momentan ist deine Klasse **`Orakel`** noch recht **statisch**. Es wäre doch toll, wenn das Orakel je nach Verwendung auch **andere Antworten** geben könnte.

Eine Möglichkeit, mehr (oder andere) Antworten bereitzustellen, ist der **Zugriff** auf das Attribut **`antworten`**.

So, wie du auf eine Methode zugreifen kannst, kannst du ja auch auf ein **Attribut zugreifen**, um es zu lesen, und genauso, um es zu **verändern**, und zwar zur **gesamten Laufzeit** des Programms (also solange das Programm läuft). So wie du ein Attribut mit

- dem **Namen** deines Objekts
- plus einem **Punkt**
- plus dem **Namen** des Attributs

lesen kannst, kannst du genauso schreibend darauf zugreifen, also dem Attribut einen neuen Wert zuweisen – natürlich erst, nachdem du ein entsprechendes Objekt erstellt hast:

```
mein_orakel.antworten = ("Tu es, Schrödinger!", "Lass es sein",
                         "Überleg dir das noch einmal",
                         "Frag doch lieber deine Freundin")
```

Schon hast du ab dem Moment der Zuweisung **ganz andere** Antworten.
Das kannst du dann gleich ausprobieren:

```
weissagung*1 = Orakel()
weissagung.ausgabe()*2
weissagung.antworten*3 = ("Tu es, Schrödinger!", "Lass es sein",
"Überleg dir das noch einmal", "Frag doch lieber deine Freundin")
weissagung.ausgabe()*4
```

*1 Vorher hieß dein Objekt **`mein_orakel`**. Hier geben wir dem Objekt einfach mal einen ganz **anderen Namen**: Der Name des Objekts muss dem Namen der Klasse nicht einmal ähnlich sein.

*2 Hier sind noch die »alten« Antworten hinterlegt.

*3 Hier weist du während der Laufzeit (also während das Programm läuft) dem Attribut ein ganz neues Tupel mit Antworten zu.

*4 Ab dem Moment der Zuweisung werden die neuen Antworten ausgegeben. Das Objekt funktioniert aber ansonsten unverändert.

```
++++++++++++++++++++
Das Orakel spricht 'Ja'
++++++++++++++++++++*2
++++++++++++++++++++++++++
Das Orakel spricht 'Lass es sein'
++++++++++++++++++++++++++*4
```

Trotz der Änderung läuft das Programm ohne Probleme weiter!

Vorsicht beim Zugriff auf Attribute!

Der direkte Zugriff auf Attribute ist ja einfach – geradezu **verführerisch** einfach. Das ist aber auch so etwas wie die **dunkle Seite** der OOP. In unserer Klasse wird erwartet, dass das Attribut **`antworten`** ein **Tupel** enthält. Du könntest dem Attribut aber auch einen String oder (noch schlimmer) eine Zahl zuweisen, was zu komischen Effekten bzw. zu einem **Fehler** führen würde! Auch benötigt das Programm bei einem Tupel mindestens zwei Elemente – sonst kann es keine neuen Antworten finden und verfängt sich in einer **Endlosschleife**!

Wenn du **Attribute** direkt veränderst und unvorsichtig bist, kann es immer passieren, dass das Programm mit den so geänderten Daten nicht mehr korrekt arbeiten kann.

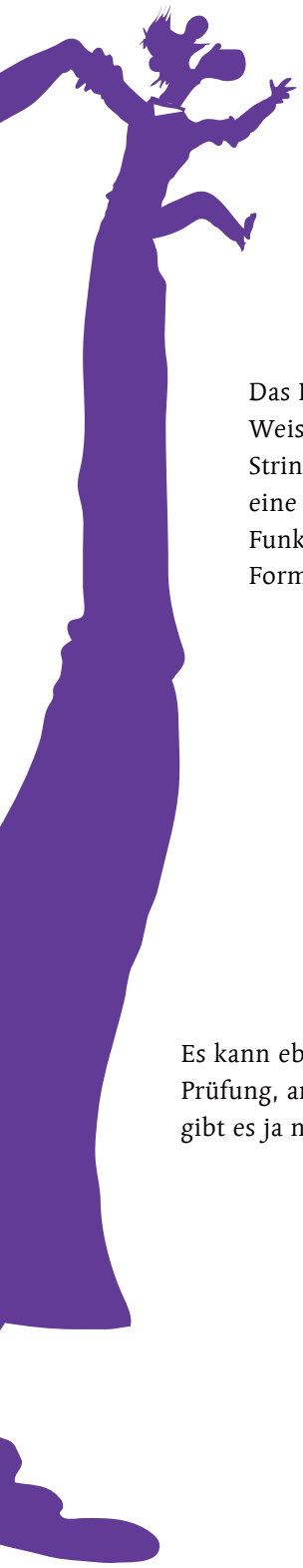
Ach, was soll denn da schon passieren?

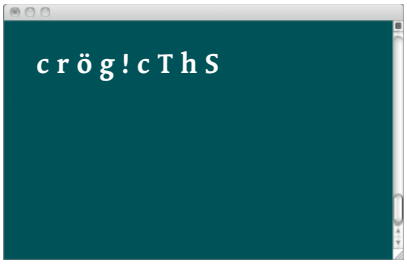
Das gleich sehen du wirst, junger Schrödinger!

[Einfache Aufgabe]
Probiere bitte einmal folgenden Code mit der ursprünglichen Klasse aus.

```
dunkles_objekt = Orakel()
dunkles_objekt.antworten = "Tu's nicht, Schrödinger!"
for i in range(9):
    print(dunkles_objekt.weissagt(), end=" ")
```

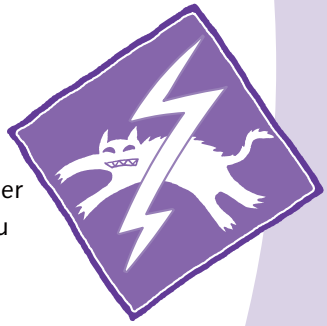
Das mach ich ganz schnell!





Das Programm geht davon aus, dass sich im Attribut **`antworten`** ein Tupel befindet. Weist du dem Attribut (beispielsweise) einen String zu, betrachtet das Programm den String als Tupel – jeder Buchstabe ist dabei ein einzelnes Element. Weist du dem Attribut eine einzelne Zahl zu, kommt es zu einem **Fehler**, da unter anderem die verwendete Funktion **`len`** nicht mit einer Zahl arbeiten kann. Und übergibst du ein Tupel in der Form **`('Ja',)`**, also mit nur **einem Element**, kommt es zu einer Endlosschleife!

[Achtung]
Natürlich kannst du Attribute direkt verändern, du solltest aber wissen (oder zumindest ausreichend testen), was du da mit deiner Zuweisung bewirkst!



Es kann eben problematisch sein, wenn **Daten direkt**, ohne die Möglichkeit einer Prüfung, an ein Programm bzw. dein Objekt übergeben werden. Aber zum Glück, gibt es ja noch andere Möglichkeiten, Daten zu übergeben!

Na, dann mal los mit den anderen Möglichkeiten!

Die Sache mit den Parametern

Die einfachste Möglichkeit ist die Übergabe von **Parametern** in eine der Methoden. Das funktioniert ganz ähnlich wie bei Funktionen:

```
class Spam:
    def verdopple_wert(self, wert):*1
        wert = wert * 2
        return wert
egg = Spam()
print(egg.verdopple_wert(42)*2)
```

*1 Auch wenn du eigene Parameter verwendest, **self** muss nach wie vor angegeben werden.

*2 Beim **Aufruf** von außen wird der Parameter **self** **nicht berücksichtigt**, darum kümmert sich Python.

[Achtung]
Beim Aufruf außerhalb der Klasse spielt der Parameter **self** keine Rolle, er wird quasi intern beim Aufruf von Python ergänzt.

Dabei bist du natürlich **nicht** auf **einen** Parameter beschränkt, sondern kannst losprogrammieren und mit den verschiedensten Parametern arbeiten.

Das könnte beispielhaft so aussehen:

```
def berechne_etwas(self, wert, multiplikator=2, trenner="~"):
```

Das ist natürlich nur ein Beispiel für die Signatur einer Methode irgendeiner Klasse.

Die möglichen Aufrufe für eine solche Methode wären dann auch wie gewohnt:

```
irgendein_objekt.berechne_etwas(42, 3, "-/-")
```

Oder etwa so:

```
irgendein_objekt.berechne_etwas(12)
```

Aber sag mal, hier könnte ich ja auch unkontrolliert Werte eingeben?

Das stimmt – noch.
Aber schließlich kannst **du** die Daten jetzt direkt nach der Übergabe überprüfen.

Vertrauen ist gut, Kontrolle besser

Du kannst jetzt **innerhalb** der Methode übergebene Werte direkt **nach der Übergabe** überprüfen und auch gleich eine Rückmeldung geben, etwa mit einer warnenden Ausgabe oder indem du **None** zurückgibst, wenn der Parameter **wert** keine Zahl ist:

```
class Spam:
  def verdopple_wert(self, wert, multiplikator=2):
      if(type(wert) != int and type(wert) != float):*1
          return None*2
      wert = wert * multiplikator
      return wert
```

*1 Hier wird der Typ der Variablen überprüft. Wenn der Wert nicht vom Typ **Integer** oder **Float** ist, dann wird es wohl keine Zahl sein …

*2 … und dann kannst du die Funktion auch gleich mit **return** wieder verlassen. Als »Fehlerhinweis« ist es durchaus üblich, **None** zurückzugeben.

Falls ein in die Methode übergebener Wert nicht korrekt ist, könntest du einen Warnhinweis ausgeben oder wie hier besser **None** (bzw. einen anderen aussagekräftigen Wert) **zurückliefern**. Dann kann **nach dem Aufruf** überprüft werden, ob die Methode korrekt abgearbeitet wurde:

```
egg = Spam()
ergebnis = egg.verdopple_wert(12.2)*1
if ergebnis is None:*2
    print("Eingabe ungültig!")
else:*2
    print(ergebnis)
```

*1 Ganz klassisch übergibst du hier den Rückgabewert, das Ergebnis der Methode, an eine Variable.

*2 Und hier reagierst du entsprechend auf den Rückgabewert.

—FÜNFZEHN—

Wenn der eigene Kopf schon raucht

Künstliche Intelligenz & Data Science

Künstliche Intelligenz ist ein ganz heißes Thema. Computer können Entscheidungen treffen und scheinbar wie von Geisterhand Ergebnisse vielleicht nicht wirklich vorhersagen, aber doch mit geradezu traumwandlerischer Sicherheit irgendwie bestimmen. Dabei gibt es die unterschiedlichsten Bereiche …

Es gibt maschinelles Lernen, Deep Learning, neuronale Netzwerke, genetische Algorithmen, Bilderkennung oder das Verständnis von Sprache. Im Kern geht es dabei um die Auswertung und Deutung von komplexen Daten und um die Simulation (oder Nachbildung) kognitiver Fähigkeiten, die für Menschen selbstverständlich, für Computer aber doch weitgehend Neuland sind.

> Dabei handelt es sich im Kern um mathematische bzw. statistische Modelle und Verfahren, mit denen zumeist genau ein Gebiet bzw. eine konkrete Aufgabe bearbeitet werden kann. Es ist also **nicht so**, dass es **eine Art** von künstlicher Intelligenz gibt, mit der dann alle Anwendungsfälle abgedeckt werden. Für jeden Teilbereich der künstlichen Intelligenz gibt es unterschiedliche mathematische Verfahren.

Beispielsweise ist die **lineare Regression** als Sonderfall der Regressionsanalyse ein beliebtes Verfahren, um maschinelles Lernen zu ermöglichen.

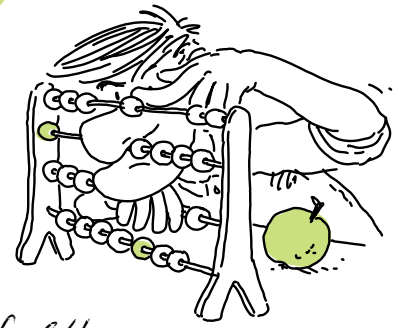
Geht das ohne diese ganzen Fachbegriffe und Mathe?

Also die verständliche Version von KI?!

Klar! Ganz um Mathematik werden wir bei diesem Thema natürlich nicht herumkommen, aber keine Sorge, die schwierigen Teile wird Python übernehmen – oder eine passende Bibliothek für künstliche Intelligenz oder Mathematik! Versuchen wir es mal ohne allzu viele Fachbegriffe und Mathematik.

Schau dir mal folgende Zahlen an:

Wert	Ergebnis
1	3
4	12
7	21
9	27
12	36
15	45
25	?



Welches Ergebnis gehört zur Zahl 25?

Das ist einfach: 75.

Die Zahl wird immer verdreifacht!

Und jetzt überleg dir mal, wie du vorgegangen bist. Du hast dir eine Zahl nach der anderen angesehen und dir überlegt, mit welcher Rechenoperation du das Ergebnis berechnen könntest. Du hast vielleicht etwas hin und her gerechnet und bist dann darauf gekommen, die Zahl jeweils **mit 3 zu multiplizieren**.

Ähnlich kann ein Computer arbeiten. Er »schaut« sich an, wie sich die Zahlen anhand zahlreicher Beispiele verändern bzw. wie sie berechnet werden. Er **lernt** aus den vorhandenen Daten – ohne die zugrunde liegende Formel zu kennen!

Aber: Ist eine Zahlenreihe **zu kurz**, wird es natürlich schwierig oder vielleicht sogar unmöglich, den nächsten Wert vorherzusagen:

Wert	Ergebnis
7	14
8	???

In diesem, zugegeben, extremen Beispiel, ist es kaum (eigentlich gar nicht) möglich, verlässlich zu sagen, wie es weitergeht: Wird zu dem jeweiligen Wert 7 hinzuaddiert? Oder wird die Zahl mal 2 genommen? Bei der **ersten Zahlenreihe** (oben) warst du in der Lage, aus den Beispielen zu **lernen**. Bei der zweiten, zu kurzen Zahlenreihe ist das nicht möglich.

[Hintergrundinfo]
Genauso ist es bei vielen Verfahren, der künstlichen Intelligenz. Je **mehr Daten** verfügbar sind, desto besser kann das jeweilige Programm lernen und dann **Vorhersagen** oder Aussagen zu einem **möglichen Ergebnis** treffen.

So geht das also: Genauso, wie **du** mithilfe von (Test-)Daten lernen konntest, um dann für neue Daten eine Vorhersage über ein mögliches Ergebnis zu machen – genauso arbeiten viele Verfahren der künstlichen Intelligenz.

Dieser (Lern-)Vorgang kann in drei Teile gegliedert werden:

1. Erst einmal werden **Daten benötigt**. Mit diesen Daten wird ein Sachverhalt beschrieben, ganz praktisch und beispielhaft. In unserem Fall ist das eine vorgegebene Zahlenreihe. Das könnten aber auch genauso irgendwelche Messdaten sein.
2. Dann folgt ein Vorgang, den man mit der **Analyse** oder noch besser mit einem **Lernvorgang** beschreiben kann: Du hast dir die Zahlenreihe angesehen und **überlegt**, mit welchen Schritten die Ergebnisse erreicht wurden. Du hast **erlernt**, was notwendig war, um die Ergebnisse zu erreichen.
3. Die **Vorhersage** ist der dritte Teil. Mit dem erlernten Wissen hast du vorhergesagt bzw. berechnet, wie das noch unbekannte Ergebnis für einen Wert aussehen müsste.

[Hintergrundinfo]
Der Computer arbeitet beim Lernvorgang (je nach Methode) etwas anders – er setzt die Werte und Ergebnisse in Relation zueinander. Versuch dir das so vorzustellen, als würdest du die Werte und Ergebnisse in ein Koordinatensystem eintragen und das Ganze zeichnerisch lösen.

Du bist in unserem recht eindeutigen Beispiel schnell auf eine einfache Formel gestoßen: `wert * 3`. Oft ist das aber eben **nicht so eindeutig** oder lässt sich eben nicht so eindeutig berechnen und es lässt sich gar **keine passende Formel** finden!

Jetzt schau dir mal diese Zahlenreihe an:

Wert	Ergebnis
1	4
3	12
5	18
10	52
12	44
15	79
20	?

Wir haben offensichtlich eine ganz ähnliche Zahlenreihe wie oben. Ein Blick auf die ersten Zahlen könnte vermuten lassen, dass der jeweilige `Wert * 4` genommen werden muss. Schaust du dir aber die übrigen Werte an, dann passt unsere eindeutige Formel nicht so recht. Damit kommen wir also nicht weiter. Die Ergebnisse haben unerklärliche Abweichungen – zumindest gegenüber unserer zuerst vermuteten Formel.

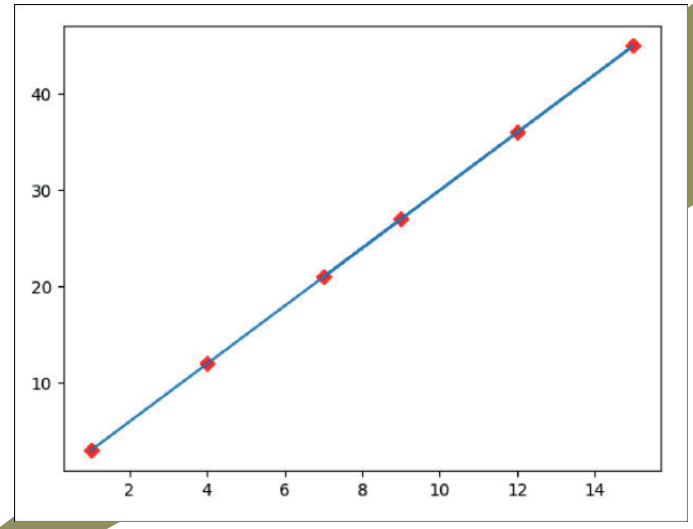
Zum Glück kann man solche Probleme aber auch ganz anders lösen. Schauen wir uns dazu das Ganze mal als Grafik an. Denn so eine Grafik sagt hier wirklich mehr als tausend Worte.

Richtige Ergebnisse – mal ganz ohne Formel

Das Ganze ist wesentlich leichter zu verstehen, wenn wir uns das einmal in einem ganz einfachen Chart ansehen. Wir nehmen dazu die Bibliothek **Matplotlib** und lassen uns die obigen Punkte in einem Koordinatensystem anzeigen.

```
import matplotlib.pyplot as plt
wert = [1, 4, 7, 9, 12, 15]
ergebnis = [3, 12, 21, 27, 36, 45]
plt.plot(wert, ergebnis, 'Db-')
plt.show()
```

Hier in aller Kürze das Programm, mit dem unsere Werte grafisch dargestellt werden. Das sind die Werte aus der ersten Tabelle.



Unsere Werte und Ergebnisse als grafische Darstellung

Wir lassen die einzelnen Punkte gleich mit einer **Linie** verbinden. Jetzt kannst du sehen, dass es selbst ohne Formel recht einfach ist, andere Punkte zu finden, die noch gar nicht als Wert mit einem Ergebnis vorliegen – zumindest, solange diese Werte den gleichen Regeln (oder Formeln) folgen:

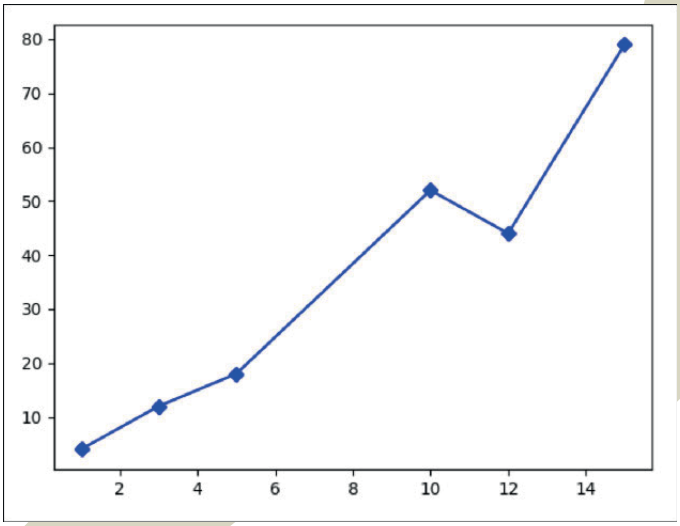
Zum Beispiel lässt sich das für den **Wert `10`** auf der **x-Achse** über die Linie ziemlich eindeutig ablesen – das **Ergebnis** ist **`30`** auf der **y-Achse**. Wenn du mit der jetzt bekannten Formel **`x * 3`** nachrechnest, kommst du natürlich zu dem gleichen Ergebnis …

Du siehst, auch **ohne eine Formel** zu kennen, könntest du dir zu beliebigen Werten das Ergebnis herleiten!

Wir haben in unserem Chart eine ganz **einfache Linie**, die unsere bekannten Punkte **direkt** verbindet.

Ganz anders ist das bei unserer zweiten Zahlenreihe!

Werfen wir mal einen Blick auf unsere zweite Zahlenreihe, indem wir sie ebenfalls mit **Matplotlib** darstellen lassen und alle Punkte genauso mit einer Linie verbinden lassen.



Das ist alles andere als eindeutig. So kommen wir wohl nicht weiter …

Du siehst sofort: So kommen wir nicht weiter.
Es ist keine gerade Linie, die wir für eine Vorhersage neuer Werte verwenden könnten.

Gehen wir mal einen Schritt weiter in die Richtung, wie Computer so etwas lösen:

KI mithilfe der linearen Regression!

Und damit wir uns gar nicht so mit der Rechnerei beschäftigen müssen, machen wir das mithilfe der Python-Bibliothek **NumPy**, die ja besonders für schnelle und umfangreiche mathematische Berechnungen geeignet ist. Damit wird so eine Regression zum Kinderspiel.

Wir haben uns ja schon mit **NumPy** beschäftigt, das sollte also nichts wirklich Neues sein.

Mit der Anweisung `pip install numpy` oder deiner Entwicklungsumgebung ist **NumPy** schnell installiert (falls du das noch nicht gemacht haben solltest). Und schon kannst du den Chart neu zeichnen lassen – samt linearer Regression:

```
import numpy as np
import matplotlib.pyplot as plt
wert = np.array([1, 3, 5, 10, 12, 15])*1)*2
ergebnis = np.array([4, 12, 18, 52, 44, 79])*1)*2
plt.plot(wert, ergebnis, 'Dr')*3
m, b = np.polyfit(wert, ergebnis, 1)*4
plt.plot(wert, m*wert + b)*5
plt.show()
```

*1 Hier sind unsere Arrays, …

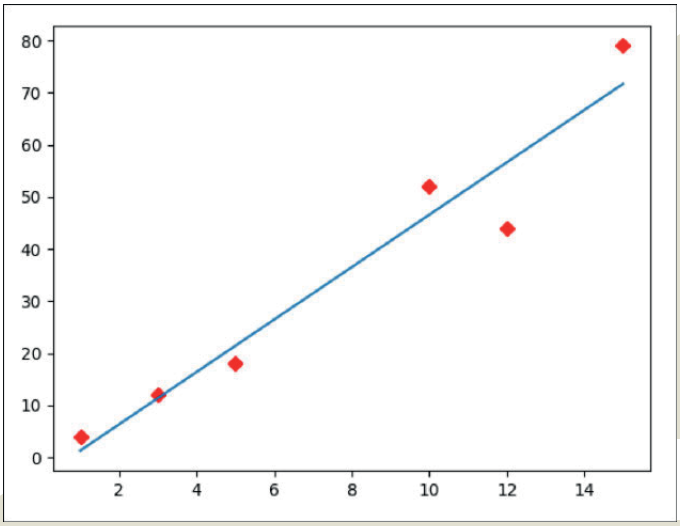
*2 … mit denen wir eindimensionale **numpy**-Arrays erzeugen.

*3 Hier lassen wir die Punkte in unser Koordinatensystem zeichnen. Bis hierhin gibt es eigentlich nichts Neues.

*4 Interessant wird es mit der Methode **polyfit**, mit der eine Kurve modelliert werden kann, die aus den übergebenen Daten generiert wird. **m** ist die Steigung, **b** der Achsenabschnitt, die wir für die Kurve benötigen, …

*5 … um hier die Kurve zeichnen zu lassen.

Dann sieht es schon ganz anders aus:



Es geht doch! Da haben wir wieder eine Linie.

Damit haben wir wieder **eine Linie**, auf der wir zu Werten die möglichen Ergebnisse ablesen könnten. Das ist natürlich etwas hemdsärmelig, aber so ungefähr kannst du dir die Vorgehensweise bei Vorhersagen (oder Lösungen) mit künstlicher Intelligenz vorstellen – wobei es auch hier natürlich die unterschiedlichsten Methoden gibt, von denen wir uns ein paar ansehen wollen.

Und jetzt alles mit echter KI

Nachdem wir uns nun unsere Werte auch wortwörtlich grafisch vor Augen geführt haben, ist es an der Zeit, die Auswertung in Python weiterzuführen.

Genau das, was du selbst gemacht hast, **diese drei Schritte** (Daten beschaffen, lernen und dann eine Vorhersage machen) werden wir jetzt in Python **programmieren**. Und wir nehmen dazu eine Bibliothek namens **Scikit-learn**. Das ist, wie der Name (zumindest ein bisschen) vermuten lässt, eine Bibliothek für **maschinelles Lernen**. Du findest sie unter der Adresse **https://scikit-learn.org**. Installieren kannst du diese Bibliothek entweder über die Entwicklungsumgebung deiner Wahl oder mit `pip` über die Kommandozeile:

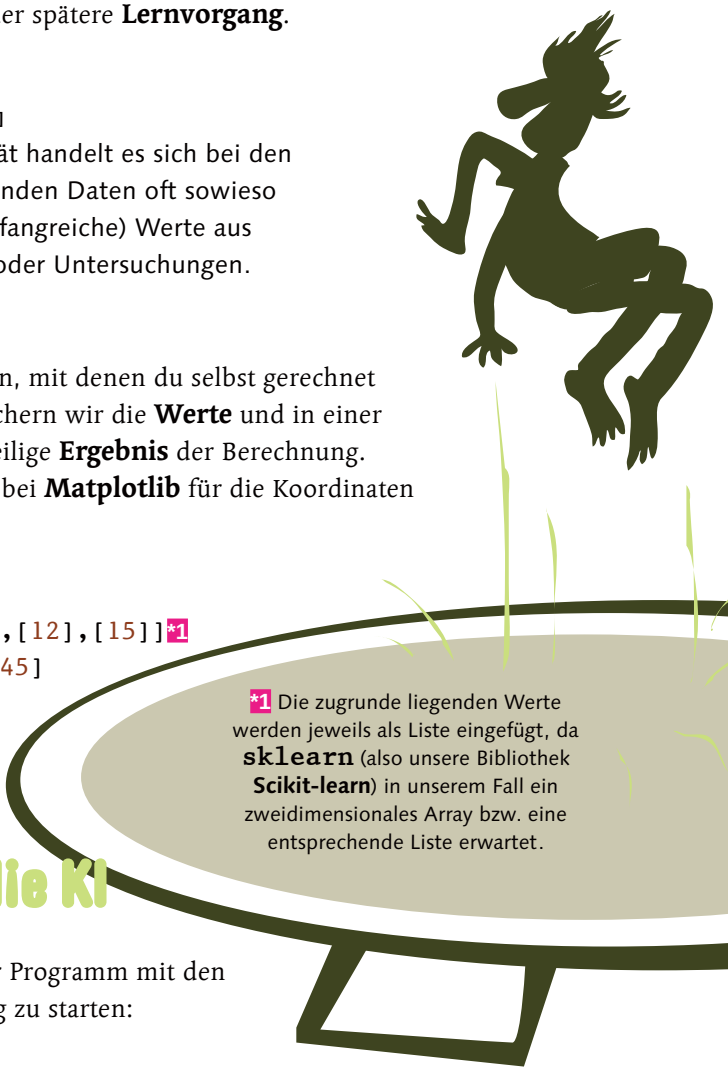
```
pip install scikit-learn
```

Als Grundlage für unser nun folgendes Programm dient uns ein Programm von **Raman Sah**, das er uns freundlicherweise zur Verfügung gestellt hat und das in seinem Blogbeitrag **https://towardsdatascience.com/simple-machine-learning-model-in-python-in-5-lines-of-code-fe03d72e78c6** nachzulesen ist.

(Trainings-)Daten braucht das Land

Natürlich könnten wir Testdaten berechnen lassen, so arbeiten zahlreiche Beispiele, mit denen KI erklärt wird. Ist eine Formel bekannt, können damit beliebig viele – besser möglichst viele – Daten berechnet werden. Denn grundsätzlich gilt: je mehr Daten, **desto besser**. Je mehr Daten zugrunde liegen, desto verlässlicher ist der spätere **Lernvorgang**.

[Hintergrundinfo]
In der Realität handelt es sich bei den zu verwendenden Daten oft sowieso um (sehr umfangreiche) Werte aus Messungen oder Untersuchungen.



Wir jedenfalls nehmen erst mal unsere obigen Daten, mit denen du selbst gerechnet hast. In einer Liste **`training_eingabe`** speichern wir die **Werte** und in einer zweiten Liste **`training_ergebnis`** das jeweilige **Ergebnis** der Berechnung. Solche zusammengehörigen Listen hast du ja schon bei **Matplotlib** für die Koordinaten der x- und y-Achsen kennengelernt.

```
training_eingabe = [[1],[4],[7],[9],[12],[15]]*1
training_ergebnis = [3,12,21,27,36,45]
```

*1 Die zugrunde liegenden Werte werden jeweils als Liste eingefügt, da **`sklearn`** (also unsere Bibliothek **Scikit-learn**) in unserem Fall ein zweidimensionales Array bzw. eine entsprechende Liste erwartet.

Nicht für die Schule lernt die KI

Jetzt ist es an der Zeit, unser Modell bzw. unser Programm mit den Testdaten zu füttern und damit den Lernvorgang zu starten:

```
from sklearn.linear_model import LinearRegression*1
vorhersage*2 = LinearRegression()
vorhersage.fit(X=training_eingabe, y=training_ergebnis)*3
```

*1 Wir importieren das Modul für die lineare Regression.

*2 Daraus erschaffen wir uns ein Objekt, mit dem wir im Folgenden weiterarbeiten werden.

*3 Der Methode **fit** übergeben wir die Werte und die zugehörigen Ergebnisse.

Auch wenn es unglaublich erscheint, damit hat unser Modell schon **alles**, was es benötigt, um zu beliebigen Werten ein passendes Ergebnis zu finden! Dabei ist die zugrunde liegende **Formel** dem Programm gar **nicht bekannt**.

Zeit, das Orakel zu befragen

Natürlich ist hier kein Orakel am Werk, aber überraschend ist es schon, wie leicht unsere Bibliothek **Scikit-learn** mit wenigen Daten (und ohne die zugrunde liegende Formel) richtige Ergebnisse liefern kann:

```
zu_testen = [[25]]*1
ergebnis = vorhersage.predict(X=zu_testen)*2
parameter = vorhersage.coef_*3
print('Ergebnis:', ergebnis)
print('Koeffizient:', parameter)*4
```

*1 Hier geben wir einen Wert an, zu dem wir das Ergebnis haben möchten. Hier die **25**, die auch wieder als zweidimensionale Liste übergeben werden muss.

*2 Diesen Wert übergeben wir der Methode **predict** und weisen das Ergebnis einer Variablen zu.

*3 Und wir wollen noch den Koeffizienten haben, also die Zahl, die unseren Wert in einer Operation verändert.

*4 Ausgeben wollen wir beides natürlich auch noch.

Und hier das Ergebnis:

```
Ergebnis: [75.]
Koeffizient: [3.]
```

Unglaublich, aber wahr. Mit nur wenigen Daten (und immer noch ohne Formel) wird nach dem Lernvorgang das richtige Ergebnis gefunden. Das kannst du gerne noch mit anderen Werten ausprobieren. Auch der Koeffizient passt!