Einführung in die Programmierung

Andreas Hildebrandt

Grundlagen

- **Programm** = Anweisungen für einen Computer
- **Software** = alle Arten von Programmen und Programmsammlungen
- In der Regel fertig geliefert, heruntergeladen, bereit zur Ausführung
- Programme haben **Namen** („Programmnamen“)
- **Programmstart** („Aufruf“) durch
  - Eingabe des Namens
  - Klick auf ein Icon
  - automatisch ohne Zutun
  - ...

Grundlagen

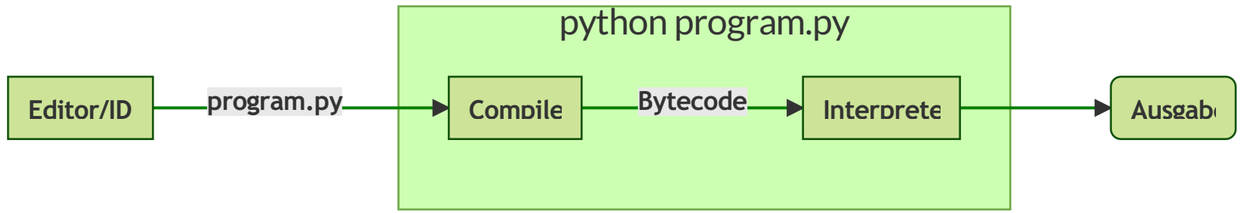
- **Softwareentwicklung** = Erstellen neuer Programme
- Einmal erstellen, oft verwenden ⇒ Aufwand lohnt sich
- Reines „Schreiben“ des Programmcodes nur Teil der tatsächlichen Arbeit
- Softwareentwicklung umfasst ...
  - Klären, welches Problem das neue Programm überhaupt lösen soll („Anforderungsdefinition“)
  - Aufbau des neuen Programms überlegen („Entwurf“)
  - Programm schreiben („Programmierung“)
  - Sicherstellen, dass das Programm zuverlässig funktioniert („Test“)

Software-Entwicklung in Python

- Programme als Text-Dateien erstellt, müssen auf `.py` enden
- Erzeugen und Bearbeiten mit beliebigem Text-Editor (Vim, Emacs, Sublime 2, Notepad, ...)
- Komfortabler: **Integrierte Entwicklungsumgebung** (Integrated Development Environment, IDE)
- In diesem Kurs verwendet: **PyCharm** von JetBrains (genaueres in den Übungen)
- **Wichtig für Python-Programmierung:** Ersetzen von Tabs durch Spaces im Editor einschalten!
  - üblicherweise schon voreingestellt bei Python-IDEs
  - muss in vielen Editoren nachgeholt werden
  - Grund für Ersetzung besprechen wir später

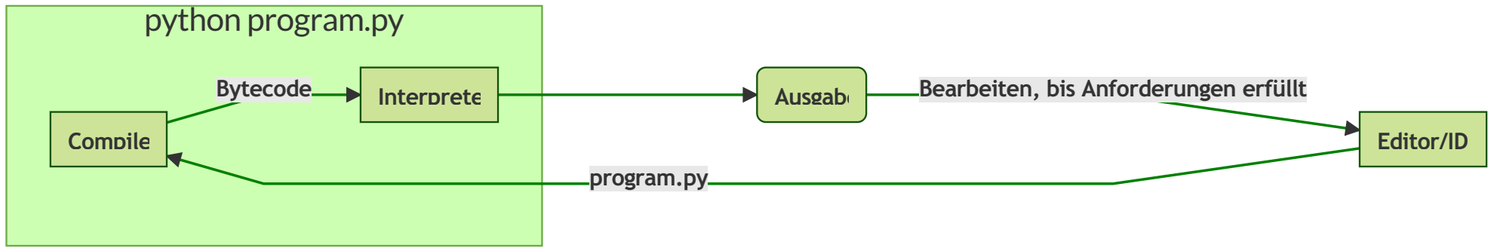
Software-Entwicklung in Python

- Ablauf bei Software-Entwicklung in Python:
  - **Programmerstellung** durch Programmierer
  - **Programmausführung** durch Nutzer



Software-Entwicklung in Python

- Typischer **Entwicklungszyklus:**
  - Erstelle Programm
  - Führe Programm aus und Teste, das korrekte/erwartete Ausgabe erzeugt wird
  - Bearbeite Programm und Teste erneut, bis Anforderungen erfüllt sind



Triviales Beispiel: Hallo Welt-Programm

- Typisches erstes Programm in neuer Programmiersprache: gebe Hello, World! aus
- Eingeführt 1974 von Brian Kernighan als Beispiel für Dokumentation der Sprache C
- Zeigt auf einfache Weise, was alles für minimales Programm in der Sprache benötigt wird
- In Python extrem einfach:

```
print("Hello, World!")
```

- **Hinweis:**
  - angezeigte Farben *nicht* Bestandteil des Programmes
  - werden vom Editor / der IDE erzeugt und angezeigt, können auch fehlen oder variieren
  - dienen nur dazu, Programm leserlicher darzustellen (**"Syntax-Highlighting"**)
  - haben keinerlei syntaktische oder semantische Bedeutung für Compiler/Interpreter

Triviales Beispiel: Hallo Welt-Programm

```
print("Hello, World!")
```

- Ausführen auf mehrere Arten möglich
- Betrachten im Moment nur:
  - Speichern obigen Programmtext (*"Quelltext"*, *"Source Code"*) in Datei hello.py
  - Ausführbare Quelltextdatei bei Python auch als **Skript** bezeichnet
  - Ausführen auf der Kommandozeile mit

```
[andreas@localhost] $ python hello.py
                     Hello, World!
```

Triviales Beispiel: Hallo Welt-Programm

- Was macht dieses Programm bei Ausführung?

```
print("Hello, World!")
```

- **Intuitive Idee:** Programm *"druckt"* den Text Hello, World!
- **Genauer:** Programm gibt bei Ausführung die Zeichenkette (*String*) "Hello, World!" aus
- Aber wohin?

Die Standardströme

- Computer-Steinzeit: Programme fest gekoppelt mit physischer Hardware für
  - **Eingabe:** Tastatur
  - **Ausgabe:** Monitor oder Drucker
- Sehr unflexibles Konzept, Programme müssen Details der Hardware kennen, angepasst werden bei Änderungen
- In den 1970er Jahren flexible Lösung, eingeführt mit Unix und C: **Standardströme**
- Betriebssystem verwaltet Ein- und Ausgabegeräte und *bindet* Strom daran
- Bindung kann jederzeit verändert werden (**"Redirection"**, **"Piping"**, siehe später)
- Standardströme **abstrahieren** von Ein- und Ausgabegeräten (**I/O**-Geräte)

Die Standardströme

- Standardströme werden automatisch ohne Zutun des Benutzers vom Betriebssystem verbunden...
- ...können aber immer noch umgeleitet werden
- Heute üblich: drei Standardströme
  - **Standardeingabe (stdin):** normalerweise verbunden mit Tastatur
  - **Standardausgabe (stdout):** normalerweise verbunden mit Terminal-Applikation (*"Kommandozeile"*), aus der Programm gestartet wurde
  - **Standardfehlerausgabe (stderr):** normalerweise verbunden mit Terminal-Applikation (*"Kommandozeile"*), aus der Programm gestartet wurde, verwendet zur Ausgabe von Fehler- und Warnmeldungen

Die Standardströme

- Weitere Möglichkeiten für I/O (z.B. Dateien, Netzwerk, Graphische Oberflächen, ...) betrachten wir später
- Derzeitiges Ausführungsmodell: Python-Programm erzeugt bei Ausführung eine Ausgabe auf stdout und/oder stderr

Die Standardströme - Redirections

- Bindung der Ströme kann einfach verändert werden
- Zwei Mechanismen:
  - **Redirections:** lenken Strom in andere Quelle / anderes Ziel um
  - **Piping:** verbinden Ausgabe eines Programms mit Eingabe eines anderen
- Betrachten zunächst nur Redirections von stdout

Die Standardströme - Redirections

- Auf Kommandozeile üblicher Betriebssysteme (Linux, MacOS, Windows) redirection einfach zu erreichen: ersetze

```
[andreas@localhost] $ | python hello.py
```

  durch

```
[andreas@localhost] $ | python hello.py > hello.txt
```

- Lenkt Standardausgabe in Datei hello.txt um:
  - Ausgabe `Hello, World!` nicht länger auf Bildschirm angezeigt
  - Statt dessen neue Datei mit Inhalt `Hello, World!` erzeugt

Die Standardströme - Redirections

- Existiert Datei bereits vor redirection wird sie überschrieben
- Alternativ:

```
[andreas@localhost] $ | python hello.py >> hello.txt
```

  - hängt an Datei an, falls bereits existent
  - erzeugt sonst neue Datei

Anatomie des Beispiels

- Schauen wir uns Hello, World! genauer an

```
print("Hello, World!")
```


JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Anatomie des Beispiels

- Schauen wir uns Hello, World! genauer an

```
print("Hello, World!")
```

- Sog. **Funktionsaufruf** der Funktion `print` (Funktionen besprechen wir später im Detail)
- `print`: Aufforderung an Interpreter/Betriebssystem, übergebene Zeichenkette auf stdout auszugeben

Anatomie des Beispiels

- Schauen wir uns Hello, World! genauer an

```
print("Hello, World!")
```

- Sog. **Parameter** der Funktion `print`
- Zeichenkette, die ausgegeben werden soll

Einführung in das Ausführungsmodell

- Python-Skripte meist erheblich komplexer als das Beispiel
- Einfachste Erweiterung: mehrere **Anweisungen** (**Statements**)
- Können jeweils in eigener Zeile (genauer: **logische** Zeile; Regeln betrachten wir noch) angehängt werden
- Werden dann von Python von oben nach unten Zeile für Zeile ausgeführt
- **Wichtig:**
  - in Python ist **Einrückung** der Zeilen entscheidend
  - Einrückung eines Statements entspricht Anzahl Leerzeichen vor seinem ersten Zeichen
  - Muss für alle Anweisungen des gleichen Blocks (genaue Definition später) **identisch** sein

Einführung in das Ausführungsmodell

- Einrückung in der Praxis manchmal problematisch bzw. unintuitiv
- Probleme vor allem bei Verwendung von **Tabulator-Zeichen** (tab, \t)
- In Python so interpretiert:
  - Tabulator-Zeichen werden durch minimale Anzahl Leerzeichen ersetzt...
  - ...die die Gesamtzahl an Leerzeichen durch 8 teilbar macht
- Zur Vermeidung von Fehlern: Tab und Space (Leerzeichen) bei Python nicht mischen
- Noch einfacher: keine Tabulator-Zeichen in Datei speichern, sondern Leerzeichen
- Gute Editoren können so eingestellt werden (z.B. vim: set et)
- In Python-IDEs meist voreingestellt

Einführung in das Ausführungsmodell

- Beispiel mit korrekter Einrückung:

```
print("Hello, World!")
print("Nice to meet you!")
print("How are you?")
```

- Ausgabe:

```
[andreas@localhost] $ python hello.py
  Hello, World!
  Nice to meet you!
  How are you?
```

Einführung in das Ausführungsmodell

- Beispiel mit inkorrekter Einrückung:

```
print("Hello, World!")
 print("Nice to meet you!")
print("How are you?")
```

```
[andreas@localhost] $ python hello.py

  File "hello.py", line 2
  print("Nice to meet you!")
  ^
IndentationError: unexpected indent
```

Einführung in das Ausführungsmodell - Layout

- Abgesehen von Zeilenanfang (und Stringliteralen, siehe später) Leerzeichen und Tabulatoren an vielen Stellen ignoriert
- Regeln in etwa:
  - Symbole dürfen nicht zerrissen werden

    ```
    pr int("Hello, World!")
    ```

    inkorrekt und nicht akzeptiert
  - *zwischen* Symbolen *dürfen* Leerzeichen stehen, werden aber nur benötigt, wenn sonst misverständlich

    ```
    print("Hello, World!")
    print ("Hello, World!")
    print( "Hello, World!" )
    ```

    alle korrekt, ergeben alle gleiche Ausgabe

Einführung in das Ausführungsmodell

- Leere Zeilen werden i.A. ignoriert (**Ausnahme:** interaktiver Modus nach multi-line statement)
- Gilt auch für Zeilen, die nur Whitespace enthalten
- Beispiel:

```
print("Hello, World!")

print("Nice to meet you!")

print("How are you?")
```

```
[andreas@localhost] $ python hello.py
  Hello, World!
  Nice to meet you!
  How are you?
```

Lexer und Parser

- Was passiert mit Quelltext bei Kompilierung?
- Erster Schritt: **lexikalische Anlyse**
  - Durchgeführt von sogenanntem **Lexer**
  - Zerlegt Quelltext zunächst in **Lexeme** - Zeichenfolgen, die für jeweils ein **syntaktisches Objekt** stehen
  - Weist dann Lexemen grammatische **Kategorien** zu - sogenannte **Token**
  - Ergebnis ist Abfolge von Lexemen/Token
- Analogie in natürlicher Sprache:
  - "Hello, World!" hat Lexeme `'Hello'`, `','`, `' '`, `'World'`, `'!'` mit zugehörigen Token (z.B.) `Wort`, `Satzzeichen`, `Leerzeichen`, `Wort`, `Satzzeichen`

Lexer und Parser

- Im Beispiel:

```
print("Hello, World!")
```

- Lexem ⇒ Token:
  - `'print'` ⇒ NAME
  - `'('` ⇒ OP
  - `'Hello, World!'` ⇒ STRING
  - `')'` ⇒ OP
  - `'\n'` ⇒ NEWLINE
  - `''` ⇒ ENDMARKER

Lexer und Parser

- Was passiert mit Quelltext bei Kompilierung?
- Zweiter Schritt: **syntaktische Anlyse**
  - Durchgeführt von sogenanntem **Parser**
    - überprüft, ob Token-Abfolge durch Python-Grammatik erlaubt ist
    - erzeugt dabei sogenannten **Parsebaum** oder **Abstract Syntax Tree (AST)** - wichtig für Kompilierung/Interpretation

Einführung in das Ausführungsmodell - Kommentare

- Quelltexte werden schnell unübersichtlich und schwer verständlich
- **Kommentare** erlauben, Erläuterungen direkt im Quelltext unterzubringen
- Gedacht nicht für Compiler oder Interpreter, sondern für **menschliche Lesern**
- Eingeleitet durch Zeichen # (genauer: außerhalb eines Stringliterals, siehe später)
- Alle weiteren Zeichen der gleichen (physikalischen) Zeile von Python komplett ignoriert
- Vollständig vom Lexer gehandhabt; erzeugen kein Token
- Beispiel:

```
# Einfaches Hello, World! - Programm für EiP-Vorlesung
print("Hello, World!")
```

Einführung in das Ausführungsmodell - Kommentare

- Kommentare oft auch verwendet, um nicht benötigten Code zu deaktivieren
- Bezeichnet man als **auskommentieren**
- Beispiel:

```
# Zweite Zeile zu Testzwecken auskommentiert
print("Hello, World!")
#print("Nice to meet you!")
print("How are you?")
```

```
[andreas@localhost] $ python hello.py
  Hello, World!
  How are you?
```

- Nützlich z.B. bei Fehlersuche

Einführung in das Ausführungsmodell - Kommentare

- Kommentare extrem wichtig zum korrekten Verständnis
- **Faustregel:** ausführlich kommentieren (wird in Übungen mit abgeprüft werden!)
- **Aber:** Kommentare eher schädlich, falls
  - Offensichtliches kommentiert wird

```
print("Hello, World!") # Gibt "Hello, World!" aus
```

  - In diesem Beispiel unschädlich...
  - ...lenkt in der Praxis aber von wichtigeren Aspekten ab

Einführung in das Ausführungsmodell - Kommentare

- Kommentare extrem wichtig zum korrekten Verständnis
- **Faustregel:** ausführlich kommentieren (wird in Übungen mit abgeprüft werden!)
- **Aber:** Kommentare eher schädlich, falls
  - Kommentar nicht (mehr?) zu Quelltext passt

```
print("Hello, World!") # Gibt "Guten Tag!" aus
```

  - Wichtige Fehlerquelle in realen Programmen!
  - Häufiges Szenario:
    - Kommentar als Dokumentation eingeführt
    - Quelltext verändert
    - Kommentar nicht angepasst

Einführung in das Ausführungsmodell - Kommentare

- Kommentare extrem wichtig zum korrekten Verständnis
- **Faustregel:** ausführlich kommentieren (wird in Übungen mit abgeprüft werden!)
- **Aber:** Kommentare eher schädlich, falls
  - Kommentar schlechten Code rechtfertigen soll

```
# Keine Ahnung was mein Code hier macht,
# aber wenn ich die Zeile lösche, stürzt er Freitags immer ab.
# Bitte nicht anfassen!
```

  - Kommt in der Praxis extrem häufig vor!
  - Verdeckt meist tieferes Problem

Einführung in das Ausführungsmodell - Kommentare

- Im Zweifelsfall: lieber **kommentieren** als Kommentare wegzulassen!
- Zu viel Kommentar schadet kaum, wenn er korrekt und aktuell ist
- Fehlende oder unzureichende Kommentare können funktionierenden Quelltext wertlos und unbrauchbar machen

Regelebenen - Syntax, Semantik und Pragmatik

- Verschiedene "Regelebenen" bei der Programmierung:
  - **Syntax:** Regeln, die Programm erfüllen muss, um kompiliert und ausgeführt werden zu können
    - Entspricht in etwa Rechtschreibung und Grammatik in natürlicher Sprache
    - Syntaxfehler entdeckt von Compiler (Lexer, Parser, teilweise Postprocessing)
  - **Semantik:** Bedeutung / Logik des Programms
    - Entspricht in etwa Inhalt in natürlicher Sprache
    - Nur teilweise überprüft von Compiler und Interpreter
  - **Pragmatik:** Üblicher Gebrauch von Stil- und Sprachelementen
    - Entspricht in etwa gutem Sprachgebrauch in natürlicher Sprache
    - Nicht überprüft von Compiler und Interpreter

Regelebenen - Syntax, Semantik und Pragmatik

- Regelverstöße haben unterschiedliche Auswirkungen:
  - **Syntaxfehler:** Syntaktisch inkorrekte Programme werden nicht ausgeführt; Compiler meldet Fehler
  - **Semantikfehler:** unterschiedliche Auswirkungen möglich:
    - werden nicht ausgeführt (entdeckt vom Compiler)
    - verhalten sich falsch, stürzen mgw. ab
  - **Fehler in der Pragmatik:** Programm schwer lesbar, umständlich, schwer zu erweitern
- Nicht erkannte Semantikfehler sind bei Weitem größtes Problem der Programmierung!
- Fehler erst erkannt, wenn Programm schon läuft (Flugzeug schon fliegt, Rakete schon gestartet ist, Toaster schon toastet, ...)

Regelebenen - Syntax, Semantik und Pragmatik

- **Beispiele:**
  - Original:

```
print("Hello, World!")
```

```
Hello, World!
```

Regelebenen - Syntax, Semantik und Pragmatik

- **Beispiele:**
  - Original:

```
print("Hello, World!")
```

  - Fehlerhaft:

```
print("Hello, World!"))
```

    - Syntaxfehler
    - gefunden von Compiler:

```
  File "<ipython-input-13-06f507e487c4>", line 1
    print("Hello, World!"))
                          ^
SyntaxError: invalid syntax
```

Regelebenen - Syntax, Semantik und Pragmatik

- **Beispiele:**
  - Fehlerhaft:

```
print(Hello)
```

    - Syntax ok, aber semantischer Fehler
    - nicht gefunden von Compiler, Laufzeitfehler bei Ausführung:

```
In [14]: print(Hello)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-b7be8ca2a54e> in <module>()
----> 1 print(Hello)

NameError: name 'Hello' is not defined
```

Regelebenen - Syntax, Semantik und Pragmatik

- **Beispiele:**
  - Fehlerhaft:

```
# Programm soll Hello, World! ausgeben
print("blörg")
```

    - Syntax ok, aber semantischer Fehler
    - nicht gefunden von Compiler, kein Laufzeitfehler bei Ausführung, aber falsches Verhalten:

```
In [15]: print("blörg")
```

```
blörg
```

Regelebenen - Syntax, Semantik und Pragmatik

- **Beispiele:**
  - "Fehlerhaft":

```
print("H", end="")
print("e", end="")
print("l", end="")
print("l", end="")
print("o", end="")
print(",", end="")
print(" ", end="")
print("W", end="")
print("o", end="")
print("r", end="")
print("l", end="")
print("d", end="")
print("!")
```

    - Syntax ok, Semantik ok, aber Verstoß gegen Pragmatik
    - Schwer zu verstehen oder erweitern, verhält sich aber korrekt:

```
Hello, World!
```

Statische und dynamische Eigenschaften

- Programmeigenschaften, die ohne Ausführung bewiesen werden können, nennt man **statisch bestimmt**
- Beispiele:
  - Syntax des Programms kann statisch analysiert werden
  - Teile der Semantik können statisch analysiert werden
- Programmeigenschaften, die nur durch Ausführung bewiesen werden können, nennt man **dynamisch bestimmt**
- Beispiel:
  - Teile der Semantik können dynamisch analysiert werden
- Programmeigenschaften, die weder statisch noch dynamisch bewiesen werden können, nennt man **unentscheidbar**
- Beispiel:
  - Frage, ob gegebenes Programm terminiert kann i.A. nicht entschieden werden
- Ob gegebene Eigenschaft statisch oder dynamisch bestimmbar ist, ist stark abhängig vom **Typsystem** der Sprache

Einführung in das Typsystem

- In Programmen sollen **Daten** eingelesen, manipuliert, analysiert, ausgegeben, ... werden
- **Datenverarbeitung** also zentrales Element der Programmierung
- Bekannt aus der Mathematik: unterschiedliche *Typen* von Daten erlauben unterschiedliche *Operationen*
- Beispiel: durch Skalare kann dividiert werden, durch Vektoren nicht
- Motiviert **Definition: Datentyp** ist Tupel aus **Wertemenge** und **Menge möglicher Operationen**

Einführung in das Typsystem

- Datentypen können auf verschiedene Weise in Programmiersprachen umgesetzt werden
- **Typsystem** der Programmiersprache daher entscheidendes Merkmal
- Zwei wichtige Klassen:
  - **Statische Typsysteme**
    - bestimmen Datentypen statisch (zur Compilezeit)
    - überprüfen korrekte Verwendung (Anwendung von Operationen aus Menge möglicher Operationen) statisch
  - **Dynamische Typsysteme**
    - bestimmen Datentypen, wenn gebraucht, dynamisch (zur Laufzeit)
    - überprüfen korrekte Verwendung dynamisch
- im Gegensatz zu vielen anderen Sprachen verwendet Python **dynamisches Typsystem**

Einführung in das Typsystem

- **Vorteil** dynamischer Typsysteme: Programmierer muss weniger Informationen angeben
- **Nachteil** dynamischer Typsysteme: Programmierer gibt weniger Informationen an...
- Verwendung des Typsystems einfacher und flexibler...
- ... aber auf Kosten präziser Information für den Compiler
- Warum ist das wichtig?
  - Compiler kann Eigenschaften ausnutzen, die er statisch beweisen kann
  - dazu braucht er verlässliche Annahmen über Datentypen
- Beispiele:
  - Compiler kann beweisen, dass alternativer Code immer gleiches Ergebnis liefert, aber schneller läuft ⇒ Programmoptimierung
  - Compiler kann beweisen, dass Daten inkorrekt verwendet werden ⇒ Fehlererkennung
- In Python oft nicht möglich!

Einführung in das Typsystem

- Mehrere Projekte kombinieren Python mit statischem Typmodell
- Wichtige Beispiele:
  - **Cython:**
    - Übersetzung in Programmiersprache C
    - Statisches Typsystem von C vor allem zu Performancezwecken genutzt
  - **mypy:**
    - Verwendet Typannotationen (verwendbar ab Python 3)
    - Statische Typprüfung zur Fehlervermeidung
    - Wollen wir später noch betrachten

Eingebaute Datentypen (built-ins)

- Python kennt mehrere im Sprachstandard verankerte (**built-in**) Datentypen
- Werden im Folgenden einige davon betrachten:
  - `int` - Ganze Zahlen (**Integers**)
  - `float` - Gleitkommazahlen (**Floating Point Numbers**)
  - `complex` - Komplexe Gleitkommazahlen (**Complex Numbers**)
  - `bool` - Wahrheitswerte (**Booleans**)
  - `str` - Zeichenketten (**Strings**)

Literale

- Oben definiert: Datentyp entspricht Wertemenge und Menge möglicher Operationen
- **Definition:** ein **Literal** eines Datentyps ist eine Python-Darstellung eines (konstanten) Elements seiner Wertemenge
- Literale numerischer Typen (`int, float, complex, ...`) heißen auch **Numerale**

Literale

- Oben definiert: Datentyp entspricht Wertemenge und Menge möglicher Operationen
- **Definition:** ein **Literal** eines Datentyps ist eine Python-Darstellung eines (konstanten) Elements seiner Wertemenge
- Literale numerischer Typen (`int, float, complex, ...`) heißen auch **Numerale**
- **Beispiele:** `int` - Ganze Zahlen
  - 1
  - 3
  - 42
  - 1234
  - ...
- **Hinweis:** im Gegensatz zur Mathematik führende `0` - Ziffern in Python nicht erlaubt! (Hintergrund: in vielen anderen Programmiersprachen für Zahlen im **Oktalsystem** verwendet, häufige Fehlerquelle!)

Literale

- Oben definiert: Datentyp entspricht Wertemenge und Menge möglicher Operationen
- **Definition:** ein **Literal** eines Datentyps ist eine Python-Darstellung eines (konstanten) Elements seiner Wertemenge
- Literale numerischer Typen (`int, float, complex, ...`) heißen auch **Numerale**
- **Beispiele:** `float` - Gleitkommazahlen mit
  - Vor- und Nachkommastellen
    - `42.0`
    - `1234.56789`
    - ...
  - Mantisse und Exponent
    - `1e1` (10)
    - `2.45e2` (245)
    - ...
  - ...

Literale

- Oben definiert: Datentyp entspricht Wertemenge und Menge möglicher Operationen
- **Definition:** ein **Literal** eines Datentyps ist eine Python-Darstellung eines (konstanten) Elements seiner Wertemenge
- Literale numerischer Typen (`int, float, complex, ...`) heißen auch **Numerale**
- **Beispiele:** `complex` - Literale nur für imaginäre Zahlen, `j` oder `J` für imaginäre Einheit i
  - `3j`
  - `42J`
  - ...

Literale

- Oben definiert: Datentyp entspricht Wertemenge und Menge möglicher Operationen
- **Definition:** ein **Literal** eines Datentyps ist eine Python-Darstellung eines (konstanten) Elements seiner Wertemenge
- Literale numerischer Typen (`int, float, complex, ...`) heißen auch **Numerale**
- **Beispiele:** `bool` - Wahrheitswerte mit nur zwei möglichen Werten
  - `True`
  - `False`

Literale

- Oben definiert: Datentyp entspricht Wertemenge und Menge möglicher Operationen
- **Definition:** ein **Literal** eines Datentyps ist eine Python-Darstellung eines (konstanten) Elements seiner Wertemenge
- Literale numerischer Typen (`int, float, complex, ...`) heißen auch **Numerale**
- **Beispiele:** `str` - Zeichenketten, umgeben von gleichartigen einfachen oder dreifachen Anführungszeichen (+ evtl. sogenannte *Modifier*, siehe später)
  - `'Hello, World!'`
  - `"Hallo, Welt!`
  - `'''Test'''`
  - `"""Einführung in die Programmierung"""`
  - …

Literale

- Literale können z.B. an Funktion `print` übergeben werden:

In [18]:
```
print(42)
print(1.2e2)
print(3.0J)
print(True)
print("Hello, World!")
```

```
42
120.0
3j
True
Hello, World!
```