Lukas Müller

Programmieren für Studierende der Naturwissenschaften

V2 – Elementare Datentypen und Kontrollstrukturen

Inhalte

- V1: Grundlagen der Programmierung
  P1: Hilfe beim Einrichten von Python an eigenen Rechnern, erste Programme ausführen
- V2: Elementare Datentypen und Kontrollstrukturen
  P2: Übungen
- V3: Aggregierte Datentypen
  P3: Übungen
- V4: Aggregierte Datentypen und Funktionen
  P4: Übungen
- V5: Testen, Fehlermeldungen und Selbsthilfe
  P5: Übungen

Inhalte

- V6: Externe Packages, Einführung NumPy und SciPy
  P6: Übungen
- V7: Externe Packages 2
  P7: Übungen
- V8: Umgang mit externen Daten und Visualisierung
  P8: Übungen
- V9: Entwurf von Algorithmen ODER Aufarbeitung besprochener Themen
  P9: Übungen, selbstständige Arbeit in Kleingruppen
- V10: Betriebssysteme (Windows, Linux, macOS) ohne Übung

Rückblick

- Eine Variable kann konzeptionell auch als ein Container im Speicher betrachtet werden, lässt sich als ein Tripel betrachten und besitzt:

- **Name (identifier):**
  ein in einem Namensraum (hier zunächst das gesamte Programm) eindeutiges Wort, unter dem die Variable im Programmtext angesprochen werden kann

- **Typ (type):**
  Zusammenfassung konkreter Wertebereiche von Variablen (z.B. ganze Zahlen) und darauf definierten Operationen zu einer Einheit

- **Wert (value):**
  „Inhalt“ der Variable (wird durch die Angabe des Typs eindeutig)
  Beispiel: DIX (Nachnamen von Otto DIX oder „römische Zahl“ = 509?)

Elementare Datentypen

- Werden von der Programmiersprache direkt unterstützt,
  - können sich von Sprache zu Sprache unterscheiden,
  - können nur einen Wert des eigenen Wertebereiches annehmen,
  - sind diskret und endlich
- Sie können häufig durch entsprechende Hardwarekomponenten effizient behandelt werden

- Ziel
  - elementare Datentypen von Python beherrschen lernen
  - ihre Eigenarten zu kennen.

Übersicht

- Numerische Datentypen
  - Ganze Zahlen (integer)
  - Gleitkommazahlen (floating point number)

- Boolescher Datentyp
- Schriftzeichen? (Strings)

- Dazu: jeweils Operatoren und Operationen
- Typumwandlung (Casten)

Grundsätzliches

- Zahlen und Schrift**zeichen** sind zweifellos elementare Datenstrukturen
- Jede übliche Programmiersprache stellt diese Datentypen als elementare Datentypen zur Verfügung
- Bei Formulierungen in Programmiersprachen versucht man dabei, die üblichen Notationen der Mathematik weitgehend beizubehalten

Integer Kodierung

- Als Basis wird das das binäre Stellenwertsystem genutzt
- Es lassen sich so positive und negative Zahlen darstellen (Zweierkomplement)
- Die Operationen auf Zahlen (Addition, Subtraktion, Multiplikation, etc.) werden durch die Hardware unterstützt.

Exkurs - Stellenwertsysteme

- „Normalerweise“ nutzen wir das Dezimalsystem (Zahlen zur Basis 10, d.h. die Ziffern von 0 bis 9 kommen vor)
  - z.B.: 23910 = 2 * 10^4 + 3 * 10^3 + 9 * 10^2 + 1 * 10^1 + 0 * 10^0
- Jede natürliche Zahl kann aber auch in einem anderen Zahlensystem dargestellt werden.
- Beispiele: 1101(2) = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 13(10)
  - 15(okt) = 13(10)
    - 13 : 8 = 1 Rest: 5
    - 1 : 8 = 0 Rest: 1
  - D(hex) = 13(10)
- Im Binärsystem wird ein Zeichen auch ein „Bit“ genannt. Dieses hat genau 2 mögliche Zustände: an oder aus (beziehungsweise 1 oder 0)
- Heute nicht dabei: Kodierung von vorzeichenbehafteten Zahlen (Zweierkomplement)

Integer Operationen

- Für Integer Operanden sind in Python ALLE gängigen Operatoren definiert (+, - etc.)
- Ganzzahldivision // und Modulo %
- Es gibt auch bitweise arbeitende Operatoren (x << y, x >> y , x & y, x ^ y, x | y)
- Logische Operatoren (and, or, not)
- Damit können beliebige Ausdrücke (Terme) wie in der Mathematik üblich erzeugt werden.
- Runde Klammern stehen zur Verfügung um die Auswertereihenfolge zu steuern.

Floating Point Numbers

- Gleitpunktzahl (auch: Gleitkommazahl, manchmal auch Fließkommazahl)
- Eine meist approximierte (also angenäherte) Kodierung einer rationalen oder reellen Zahl in einer festgelegten Anzahl von Bits (meist 32, 64, seltener 16, 128 oder gar 256 Bits)
- Die Menge der Gleitkommazahlen ist eine endliche Teilmenge der rationalen Zahlen, meist erweitert um einige Spezialelemente (+Unendlich, –Unendlich, NaN (="Not A Number "), –0, usw.)
- Frage: Warum nur Approximation? Ist nicht jede beliebige Zahl so darstellbar?

Floating Point Numbers – Darstellungsprobleme

- Begrenzung der darstellbaren Zahlen durch Speicherplatz, der pro Zahl zur Verfügung steht.

- Beispiel:
  - Ich möchte die Zahl 0,2356 abbilden
  - Angenommen, ich habe nur maximal 2 Nachkommastellen zur Verfügung
  - Keine exakte Darstellung möglich!
  - Das Beste, was möglich ist, wäre 0,24

Floating Point Numbers – Darstellungsprobleme

- Vorab: Warum ist das überhaupt wichtig? Wir wollen doch nur programmieren.
  - Antwort: Fehlerfortpflanzung durch Runden!

- Wie werden Gleitkommazahlen nun konkret am Rechner dargestellt?
  - Mehr Informationen darüber gibt es in unserer Veranstaltung GPR.

  - Gegenüber einer Integerdarstellung kann mit Gleitkommazahlen bei gleichem Speicherplatzbedarf ein viel größerer Wertebereich abgedeckt werden.
  - Beispiel:
    - 32 Bit Zweierkomplement: $-2{,}147 \cdot 10^9 \leq z \leq 2{,}147 \cdot 10^9$
    - 32 Bit Gleitpunktzahl (IEEE 754): $-3{,}403 \cdot 10^{38} \leq z \leq 3{,}403 \cdot 10^{38}$

Float - Operatoren

- Die „normalen“ Operationen :`+, -, *, /, **`
- `pow(X,Y[,Z]):` X zur Potenz Y [moduloZ]
- `round(X [, N]):` Liefert ein float, gerundet auf N Dezimalstellen nach dem Komma. Default N = 0.

- Auch mit floats funktionieren:
  - Die ganzzahlige Division X // Y → Liefert das „ganzzahlige“ Ergebnis einer Division
  - Der Modulo Operator X % Y → Liefert den „Nachkommateil“ einer Division

Boolscher Datentyp (Wahrheitswerte)

- Das kleinste Speicherelement eines Computers ist eine Speicherstelle, ein „Bit“, deren Wertebereich durch zwei Zustände 0 und 1 gegeben ist.
- Im Folgenden bezeichnen wir den Wert 0 als False und 1 als True
- Der zugehörige Datentyp wird als „Boolesch“ (nach George Boole), oder Englisch „Boolean“ bezeichnen
- Besonderheiten von Python: Jede Variable (jeder Ausdruck) kann boolesch „interpretiert“ werden
  - False für numerische Datentypen, wenn der Wert 0 vorliegt
  - False für String Datentypen, mit Länge 0
  - True für numerische Datentypen, wenn ein Wert ≠ 0 vorliegt
  - True String (sequentiellen) Datentypen mit Länge ≠ 0 hat
  - Das ist vielfach bequem, hat aber verschiedene Konsequenzen

Boolscher Datentyp (Wahrheitswerte) - Operatoren

Logische Operatoren: `and, or, not`

Boolscher Datentyp (Wahrheitswerte) – Vorsicht

```
Python 3.5.3 (default, Apr 22 2017, 00:00:00)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> a = True
>>> b = False
>>> a and b
False
>>> a or b
True
>>> a + b
1
>>> a + a
2
>>> b - a
-1
>>> type(a)
<class 'bool'>
>>> type(a-b)
<class 'int'>
```

Boolscher Datentyp (Wahrheitswerte) - Vergleichsoperatoren

- … (<, >, <=, >=, !=, ==) liefern True und False als Ergebnis

```
Python 3.5.3 (default, Apr 22 2017, 00:00:00)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more info
>>> a = 1
>>> b = 2
>>> c = 3
>>> a < b
True
>>> a > b
False
>>> a <= b
True
>>> a + b == c
True
>>> a != b
True
>>>
```

Boolscher Datentyp (Wahrheitswerte) - Vergleichsoperatoren

- Neben den eben genannten gibt es außerdem:
  - is
  - is not
- Diese Vergleichen auf Identität id() (also ob die selbe Speicherstelle angesprochen wird!). In der Übung testen!

```
Python 3.5.3 (default, Apr 22 2017, 00:00:00)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more information.
>>> x = [1,2,3]
>>> y = [1,2,3]
>>> x == y
True
>>> x is y
False
>>>
```

Strings

- Zeichenketten werden in (obenstehenden) "Anführungszeichen" geschrieben, gleichgültig ob einfache ' oder doppelte "

- Hier erleben viele Programmierer*innen einmal Überraschungen, weil hier ein „Chaos“ herrscht.

- Welches Chaos genau?
  - Es gibt noch diverse Varianten des Apostrophes, siehe z.B. in der Wikipedia, und andere Sonderzeichen
  - Die Programmiersprachen-Editoren (z.B. Idle) machen das natürlich immer richtig
  - Aber Vorsicht, wenn man Code aus Word, PowerPoint, o.ä. **übernimmt**!

String - Operatoren

- Die Operatoren + und * funktionieren auch bei Strings, haben aber eine besondere Bedeutung:

```
>>> s = "Hallo"
>>> t = "Welt"
>>> s + t
'HalloWelt'
>>> s*3
'HalloHalloHallo'
>>>
```

- Auch die Vergleichsoperatoren (>, <, >=, <=, ==, !=, is, is not)

  - Achtung! <, >, <=, >= ordnen die Zeichen (einschließlich Ziffern, Leerzeichen, Satz-und Sonderzeichen) nach dem Zahlenwert der dem Codepoint im Unicode entspricht, sodass bspw. alle lateinischen Großbuchstaben vor dem kleinen „a“ eingeordnet werden

Typumwandlung

- In Python lassen sich Datentypen ineinander umwandeln (das sogenannte Casten)

- Was macht Python bei
  - int(3.1)
  - int('3.1')
  - int(False)

Kontrollstrukturen

Kontrollstrukturen

- Fallunterscheidung, Iteration und Rekursion sind als grundlegende mathematische Lösungsmethoden zu erfassen
- Hierzu gehören insbesondere die verschiedenen Ausprägungen von Fallunterscheidungen und Schleifen für die Iteration
- Kontrollstrukturen aller moderner Programmiersprachen sind ähnlich aufgebaut

Rekursion ganz kurz

- Wer Rekursion verstehen will, muss vorher Rekursion verstehen
  - Teillösung bei jedem Schritt
  - …um sich selbst aufzurufen und eine weitere Teillösung zu produzieren.
  - Geeignete Abbruchbedingung erforderlich!
- Berechne „n!“

Rekursion ganz kurz

- Wer Rekursion verstehen will, muss vorher Rekursion verstehen
  - Teillösung bei jedem Schritt
  - …um sich selbst aufzurufen und eine weitere Teillösung zu produzieren.
  - Geeignete Abbruchbedingung erforderlich!
- Berechne „n!“

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
print(„n! für n=10 beträgt: “, factorial(10))
```

Übersicht

- Fallunterscheidungen:
  - Prinzipien
  - Graphische Repräsentationen
  - Realisierung in Python: if–elif–else

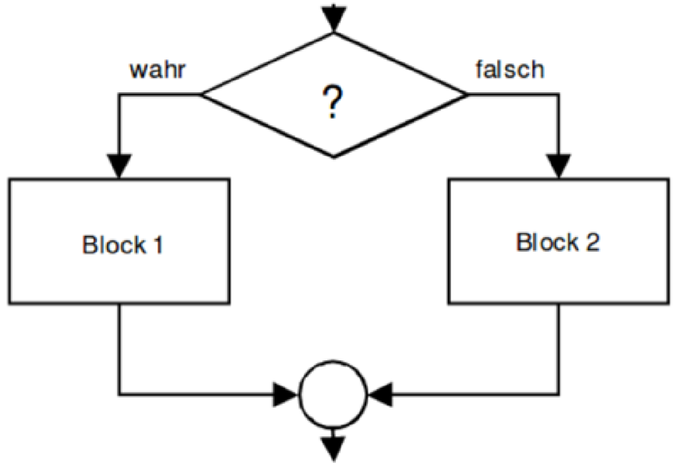
- Iterative Grundstrukturen
  - Prinzipien
  - Schleifen: Realisierungsformen der Iteration
  - Realisierungen in Python: for und while; break–continue

Verzweigung (Prinzip)

$$fib\ n = \begin{cases} n = 0 & 1 \\ n = 1 & 1 \\ sonst & fib\ (n-1) + fib\ (n-2) \end{cases}$$

Eine Verzweigung ist in allen Programmiersprachen ähnlich wie dies umgesetzt:

Verallgemeinert:
if <Bedingung>
then <Aktionsfolge>
else <Alternative Aktion>
end if



Verzweigung – in Python

```
if expression:
    suite
elif expression:
    suite
elif expression:
    suite
...
else:
    suite
```

- Wichtig:
  - Doppelpunkt nicht vergessen
  - 4 Leerzeichen als Einrückung
  - Entwicklungsumgebungen akzeptieren auch TAB (aber Vorsicht beim kopieren aus anderen Quellen!)
  - „expression“ muss zu einem Boolean ausgewertet werden
  - „suite“ ist eine beliebige Anweisung

Beispiel in Python

```
a = 3
b = 2
if a < b:
    pass
elif a == b:
    a += 1
else: a -= 1
print(a)
```

Iterative Grundstrukturen – Schleifen

- Iteration ist eine Methode, sich der Lösung eines „Rechenproblems“ schrittweise, aber zielgerichtet anzunähern
- Sie besteht in der wiederholten Anwendung desselben Rechenverfahrens
- Oft iteriert man mit Rückkopplung: Die Ergebnisse eines Iterationsschrittes (oder alle bisher erzielten Ergebnisse) werden als Ausgangswerte des jeweils nächsten Schrittes genommen

Iteration - Beispiele

- Summe:

$$a = \sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a\_n$$

- Produkt:

$$a = \prod_{i=1}^{n} a_i = a_0 + a_1 + \cdots + a\_n$$

Schleifen

- In Programmiersprachen werden iterative Lösungen beider Art durch Schleifen realisiert
- Eine Schleife wiederholt einen Teil des Codes, den so genannten Schleifenrumpf oder Schleifenkörper so lange, bis eine Abbruchbedingung eintritt
- Schleifen, die ihre Abbruchbedingung niemals erreichen oder Schleifen, die keine Abbruchbedingungen haben, nennen wir Endlosschleifen

Schleifenarten (Grundformen)

- **Die kopfgesteuerte oder vorprüfende Schleife:**
  - zuerst wird die Abbruchbedingung geprüft wird, bevor der Schleifenrumpf durchlaufen wird (meist durch das Schlüsselwort WHILE (=solange-bis) angezeigt.
- **Die fußgesteuerte oder nachprüfende Schleife:**
  - erst nach dem Durchlauf des Schleifenrumpfes wird die Abbruchbedingung überprüft z.B. durch ein Konstrukt REPEAT-UNTIL (=wiederholen-bis).
- **Die Zählschleife:** eine Sonderform der kopfgesteuerten Schleife, meist als FOR(=für)-Schleife implementiert.
- **Die foreach-Schleife:** eine Sonderform für Sequenzdatentypen: Meint ”für jedes Element in der Sequenz führe den Block genau einmal aus“.

Das Abbrechen von Schleifen

- Um eine Schleife ohne Erreichen der Bedingung (expression == False) zu beenden, verwendet man die break-Anweisung
- Um in die nächste Schleifeniteration zu springen (wobei der Rest des Schleifenrumpfes übersprungen wird) verwendet man die continue-Anweisung
- Nicht immer einfach zu verstehen. Experimentieren Sie mit vielen print()!

Beispiele - Break & continue

temp.py - /Users/alexanderwolodkin/Documents/temp.py (3.

```
search_for = eval(input("Geben Sie eine Zahl ein: "))
source = [1, 3, 14, 42]
test = 0

while test < len(source):
    if source[test] == search_for:
        print(search_for, 'liegt  in der Liste "source"')
        break
    print("test =", test, "source[test] =", source[test])
    test +=1

print("Schleife beendet")
```

Python 3.9.0 Shell

```
Geben Sie eine Zahl ein: 14
test = 0 source[test] = 1
test = 1 source[test] = 3
14 liegt  in der Liste "source"
Schleife beendet
```

temp.py - /Users/ale:

```
for i in range(5):
    if i == 2 or i == 3: continue
    print(i)

print("Schleife beendet")
```

```
>>>
============== RESTART: /Users/ale
0
1
4
Schleife beendet
>>>
```