Scripting und Algorithmen

Python I

Harald Lampesberger

FH OÖ, Department für Sichere Informationssysteme
SAL2VO, SS23, Version: 27. Februar 2023

Skriptsprachen

Primärer Einsatzzweck: Automatisierung

- zB Editor, Job Control, Graphical User Interface (GUI), Glue Code
- Bash, PowerShell, Javascript, Perl, Lua, VBA, ..., **Python**

Typische Eigenschaften von Skriptsprachen

- Interpreter, kein Compiler: dh nicht/weniger optimiert, eval-Support
- Automatische Speicherverwaltung: dh Garbage Collector
- Dynamische Typisierung: dh keine Deklaration von Variablentypen
- Imperativ mit funktionalen Sprachelementen

Python

Sehr **intuitive** Programmiersprache

- Guter Python-Code liest sich fast wie in Text
- Daher für Algorithmenentwurf gut geeignet

Hat out-of-the-box sehr mächtige Datentypen

- `bool`, `int`, `float`, `complex` ...
- Sequentielle Datenstrukturen: `tuple`, `list`, `str`, `bytes` ...
- Hash-Datenstrukturen: `set`, `dict` ...

Mächtige Standardbibliotheken

- `itertools` für Kombinatorik
- `array`, `collections` für Datenstrukturen

Großes Ökosystem an Bibliotheken

Interpreter und Versionen

Python benötigt einen Interpreter

- Referenz-Interpreter (CPython): Python http://python.org
- Jython: Java Runtime Python Interpreter
- IronPython: .NET Python Interpreter
- Pypy: Just-in-Time Compiler für Sprach-Subset
- ...

Python-Sprache existiert in zwei Versionen

- Python 2.x: alte Syntax
- **Python 3.x**: neue Syntax, nicht abwärtskompatibel

Entwicklungsumgebung

1. Anaconda Python-Distribution
   - https://www.anaconda.com/products/individual
   - Für alle Plattformen verfügbar

2. Visual Studio Code
   - Extensions: Python, Pylint, Pylance

Mehr dazu in der Übung

Everything is an Object!

Eigenschaften von Python

Imperativ (vgl. C/C++/Java...) mit funktionalen Elementen

Objektorientiert

- **Everything is an Object**
- Literale (1, 2.3, "Hello", ...) erzeugen Objekte
- Instanziierung einer Klasse erzeugt ein Objekt
- Funktionen sind Objekte
- Klassen sind Objekte (!!)

Starkes, dynamisches Typsystem

- Stark: Jedes Objekt hat ausnahmslos einen Typ
- Variablen referenzieren (zeigen auf) Objekte
- Dynamisch: Typ einer Variable steht erst zur Laufzeit fest

Ausführung

REPL: Hello World

- Python-Interpreter ist eine interaktive REPL-Shell
  - Read–eval–print loop
  - “interactive mode”

```
(base) C:\>python
Python 3.7.6 (default, Jan 8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
```

Mehr REPL

- Letztes Ergebnisobjekt in REPL immer in Variable `_`

```
>>> 6000 + 4523.5 + 134.12
10657.62
>>> _ + 8192.32
18849.940000000002
```

- Zur Vollständigkeit: REPL-Arbeiten ist mit `ipython` komfortabler
  - Sessions, Autovervollständigung, etc
  - Jupyter ist eine Client-Server-Architektur für eine REPL

Als Programm

- Programm: `helloworld.py`

```
print("hello world!")
```

- Output durch Kommando: `python helloworld.py`

```
hello world!
```

Syntax und Semantik

Unser erstes Programm

- Programm: `principal.py`

```
principal = 1000    # initial capital
rate = 0.05         # interest
numyears = 5        # number of years
year = 1
while year <= numyears:
  principal = principal * (1 + rate)
  print("year", year, "principal", principal)
  year += 1
```

- Output: `python principal.py`

```
year 1 principal 1050.0
year 2 principal 1102.5
year 3 principal 1157.625
year 4 principal 1215.5062500000001
year 5 principal 1276.2815625000003
```

Syntax

Kommentare

- Zeile ab # auskommentiert
- Multi-line String mit `"""..."""`

Blöcke durch Einrücken → keine Klammern

- Tabulator ≠ Leerzeichen!
- Am besten Leerzeichen (IDE/Editor konfigurieren)

Grundsätzlich ein Statement pro Zeile, kein Strichpunkt

- Aber mehrere Statements pro Zeile durch Strichpunkt getrennt möglich
- `a = 1; b = 1; c = a + b`
- Zwecks Lesbarkeit bitte vermeiden

Reservierte Namen

Die folgenden Bezeichner sind reserviert:

False	await	**else**	**import**	**pass**
None	**break**	**except**	**in**	**raise**
True	**class**	**finally**	**is**	**return**
and	**continue**	**for**	**lambda**	**try**
as	**def**	**from**	nonlocal	**while**
assert	**del**	**global**	**not**	with
async	**elif**	**if**	**or**	**yield**

Variablen

Unterschied zwischen C und Python

- in C/C++: Deklaration (`int a = 1;`) reserviert Speicher für Variable abhängig vom Typ
- in Python: Variable ist ein „Label“ für ein Objekt (vgl. Zeiger)
- Typen von Python-Variablen werden nicht deklariert

```
a = 1
```


```
a = 2
```


```
b = a
```

http://foobarnbaz.com/2012/07/08/understanding-python-variables/

Numerische Datentypen

Python unterstützt **`int`**, **`float`**, **`complex`**

- Integers haben unbegrenzte Präzision
- Gleitkomma verwenden normalerweise IEEE 754 double

Zahlenliterale instanzieren Zahlenobjekte

- `123` wird zu einem int-Objekt
- `123.0` wird zu einem float-Objekt
- `12.3+2j` wird zu einem complex-Objekt

`bool` ist ein Spezialfall von **`int`**

- Konstanten `True` und `False` entsprechen 1 und 0

Arithmetik

Summe: `x + y`
Differenz: `x - y`
Produkt: `x * y`
Division: `x / y`
Ganzzahl-Division: `x // y`
Modulo: `x % y`
Negation: `-x`
Tut eigentlich nichts: `+x`

Absolutwert: **`abs`**`(x)`
Nach int konvertieren: **`int`**`(x)`
Nach float konvertieren: **`float`**`(x)`
Komplexe Zahl: **`complex`**`(real, imag)`
Quotient+Rest: **`divmod`**`(x, y)`
x^y Variante 1: **`pow`**`(x, y)`
x^y Variante 2: `x ** y`

Mehr Details: https://docs.python.org/3/library/stdtypes.html

Binär, Hex, Bitweise Operatoren

Binärdarstellung

- Int zu Bit-String: `bin(15)`
  Wird zu `'0b1111'`
- Bits zu Int mit 0b-Präfix: `0b1111`
  Wird zu `15`

Hexdarstellung

- Int zu Hex-String: `hex(15)`
  Wird zu `'0xf'`
- Hex zu Int mit 0x-Präfix: `0x12`
  Wird zu `18`

Mit `int` ist folgendes möglich

- Bitweises Oder: `x | y`
- Bitweises Xor: `x ^ y`
- Bitweises Und: `x & y`
- Leftshift um n Bits: `x << n`
- Rightshift um n Bits: `x >> n`
- Negation $\sim$`x`

Vorsicht!

- Intern sind Integers unbegrenzt präzise und vorzeichenbehaftet; insb Negation verhält sich dadurch eigenartig

Wertzuweisungsoperatoren

Operatoren für Wertzuweisung

- Der Klassiker: =
- Compound-Operatoren für Arithmetik: `+=`, `-=`, `*=`, `/=`, `%=`, `//=`, `**=`
  zB `i += 1`
- Compound-Operatoren für Bitweise Operationen: `&=`, `|=`, `^=`, `>>=`, `<<=`

Python kann keine pre- oder postinkrement/dekrement Operatoren wie `i++` oder `--j`

Boolesche Operatoren

Für Verzweigungen und Schleifen müssen wir wissen, wann in Python ein Ausdruck logisch falsch ist. Welche elementaren Ausdrücke sind logisch falsch?

- Konstante `False` und Konstante `None` (vgl. NULL-Zeiger)
- Der Wert 0 bei numerischen Typen (`0, 0.0, 0+0j`)
- Wenn die Länge eines Objektes `len(x)` gleich 0 ist (vgl. leere Liste)

Boolesche Operatoren

- Negation: `not x`
- Und-Verknüpfung: `x and y`
- Oder-Verknüpfung: `x or y`

Vergleichsoperatoren

Vergleichsoperatoren für boolesche Ausdrücke

- Die üblichen Operatoren: `<`, `>`, `<=`, `>=`
- Gleichwertig: `==`
- Ungleichwertig: `!=`
- Gleiche Identität: `x is y`
- Nicht gleiche Identität: `x is not y`

Äquivalenz == vs. Identität **`is`**

- Bei numerischen Typen verhalten sich die Operatoren gleich
- Für Objektorientierung relevant

Bedingung

Syntax: **if** boolescher Ausruck**:**

- Eingerückter Block wird ausgeführt, wenn der boolesche Ausdruck logisch wahr auswertet
- Else-Zweig mittels **else:**
- Unterscheidung zu C: für „else if“ gibt es ein eigenes Sprachkonstrukt **elif**

Beispiel

```
if a or b or not c:
  pass      # noop, no operation
elif a and b:
  d = not b
else:
  print("no condition")
```

Schleifen I/II

Python unterstützt while- und for-Schleifen

- Es gibt kein do..while in Python

while boolescher Ausdruck:

- Eingerückter Block wird so lange ausgeführt, bis der boolesche Ausdruck logisch falsch wird

```
i = 3
while i > 0:
  if a:
    continue
  elif b:
    break
  else:
    print("loop...")
  i -= 1
```

continue im Block springt hoch zur Bedingungsprüfung, **break** im Block springt zum nächsten Statement nach dem Block

Schleifen II/II

for-Schleifen sind anders

- Python nutzt for-Schleifen ausschließlich zum Iterieren über Datenstrukturen
- Für numerische Indizes kann man **`range`** nutzen

`for` element **`in`** iterable:

- Eingerückter Block wird mit jedem Element aus iterable durchlaufen
- **`continue`** und **`break`** sind möglich

Beispiele

```
for char in "Hello World":
  print(char)

for i in range(10):
  print(i)    # Gibt 0..9 aus
```

Funktionen

Funktionsdefinition mittels **def**

- Im folgenden Beispiel wird ein Funktionsobjekt mit Label add erzeugt
- Variablen im Block sind lokal, Variablen aus übergeordneten Blöcken sind sichtbar
- Der Bezeichner add selbst ist wie eine Variable
- Call-Operator: (argumente)
- Ohne **return** kommt immer None zurück

```
def add(a, b):
  return a + b

add(1, 2)                # returns 3
add("Hello", "World")    # returns "HelloWorld"
```

Default-Argumente werden in der Funktionssignatur deklariert

```
def connect(host, port, timeout=300):
  pass
connect('python.org', 80)
```

Lineare Datenstrukturen

Überblick

Speicher in Computersystemen ist linear organisiert

- Linearer Adressraum, logische Blockadressen, Magnetband, ...

Grundlegende Konzepte

- Array: Elemente gleichen Typs
- Liste: einfach, doppelt verkettet
- Stack: last in first first out (LIFO)
- Queue: first in first out (FIFO)
- Priority Queue: Elemente sind nach Priorität sortiert

Grundlegende Operationen

Array

- Lese und schreibe Element an Position

Liste

- Lese, ersetze, lösche Element; Element einfügen oder anhängen

Stack

- Push / pop Element

Queue

- Enqueue / dequeue Element
- Bei Priority Queue ist enqueue eines Elementes mit einer gegebenen Priorität intelligenter und die Queue zu jedem Zeitpunkt nach absteigender Priorität sortiert

Lineare Datenstrukturen mit Python

tuple I/II

Ein Tupel (**tuple**) ist eine geordnete unveränderliche Auflistung von Objekten

- Zentrales Konzept von Python
- Typen in einem Tupel können beliebig gemischt sein; Duplikate erlaubt

```
stock = ('GOOG', 100, 490.10)
address = ('www.python.org', 80)
```

Leerer Tupel, Tupel mit einem Element

```
a = tuple()     # tuple-Konstruktor ohne Argumente
a = ()          # syntactic sugar
b = (item,)
c = item,       # syntactic sugar, c == b
```

tuple II/II

Zugriff auf Elemente mittels Subscript-Operator []

- Indizierung beginnt wie in C-Feldern bei 0
- Tupel haben eine Länge, die mit **len**

```
stock[0]      # Wert 'GOOG'
len(stock)    # returns 3
```

Auspacken in Variablen

```
name, shares, price = stock
```

```
def div_mit_rest(x, y):
  quot = x // y
  rem = x % y
  return quot, rem  # Tupel als return, coole Sache

q, r = div_mit_rest(x, y)   # automatisch entpackt
```

list I/II

Python's **list** ist eine geordnete dynamische Liste

- Listenobjekte sind „mutable", dh sie dürfen sich verändern
- Intern ist es ein dynamisches Array, aber das ist uns vorerst egal
- Typen können in einer Liste beliebig gemischt sein; Duplikate erlaubt
- Eckige statt runde Klammern im Vgl. zu Tupel

```
elements = ["SAL2VO", 123, 12.3, "SAL2UE"]
leere_liste = list()      # Constructor
auch_leer = []
```

Listen haben eine Länge

```
len(elements)   # returns 4
```

Iteration mit for-Schleife (funktioniert auch bei Tupel)

```
for e in elements:
  print(e)
```

list II/II

Liste durchsuchen

```
if 123 in elements:
  print("found!")
elements.index(123)   # returns index oder ValueError
```

Elemente verändern und einfügen

```
elements[1] = 234
elements.append("nice")
elements.insert(1, 45.0)
elements.extend(stocks)   # gleich zu: elements += stocks
a = [1, 2, 3] + [4, 5]    # Concat
```

Elemente entnehmen oder löschen

```
del(elements[1])
last_element = elements.pop()
```

Stack mit list

Unsere erste Datenstruktur

- Brauchen wir später für Bäume und Graphen

```
stack = list()        # []
stack.append(2)       # [2]
stack.append(3)       # [2, 3]
stack.append(1)       # [2, 3, 1]

stack.pop()           # 1
while stack:
  stack.pop()         # 3, 2
```

str I/III

Python's **str** ist eine unveränderliche Liste von Unicode-Zeichen

- Es gibt keinen Character-Typ in Python
- Strings können auf mehrere Varianten deklariert werden

```
leerer_string = ""
auch_leer = str()
variante1 = "Hello 'Python' World"
variante2 = 'String mit "einfachen" Anfz'
variante3 = """Multi-line 'string', der
linefeeds versteht; Anwendung auch als Kommentar"""
```

Escaping von Sonderzeichen mittels Blackslash

- Ähnlich wie in C, zB `"Hello\nWorld"`
- Raw String mit Präfix r deaktiviert Escaping
- zB `r"drei\backslashes\im\string"`

str II/III

str-Objekte unterstützen viele Methoden

- Siehe https://docs.python.org/3/library/stdtypes.html#string-methods
- Wie Tupel sind Strings „immutable“, dh sie verändern sich zur Laufzeit nicht
- Jede Operation returned ein neues str-Objekt

```
s = "Hello\nWorld"
s.lower()            # hello\nworld
s.upper()            # HELLO\nWORLD
s.count("l")         # 3
s.endswith("ld")     # True
s.index("W")         # 6
s.isalpha()          # False (wegen dem linefeed)
s.lstrip("He")       # 'llo\nWorld'
s.rstrip("rld")      # 'Hello\nWo'
s.splitlines()       # ['Hello', 'World']
s + "!!!"            # 'Hello\nWorld!!!'
```

str III/III

Zeichenweise Iteration mit for-Schleife

```
for c in "Hello\nWorld":
  print(c)
```

Typumwandlungen

- Numerische Typen können mit Konstruktor **str()** in String umgewandelt werden
- Strings können in numerische Typen mit jeweiligen Konstruktoren geparsed werden
- Unicode-Codepoint ↔ Unicode-Zeichen mittels **ord()**

```
summe = str(42) + str(24)        # '4224'
summe = int("42") + int("24")    # 66
```

Concat ist langsam; besser join

```
lst = ["strings", "to", "be", "joined"]
result = '/'.join(lst)           # 'strings/to/be/joined'
```

Slicing

Subscript-Operator für lineare Strukturen kann noch viel mehr

- `[start:stop:step]`
- Ohne Doppelpunkte ist es indizierter Zugriff
- Negative Indizes zählen von Ende in Richtung Anfang

```
s = "Hello World!"
l = [2, 3, 5, 7, 11, 13, 17]
s[-1]               # '!'
l[-2]               # 13
s[2:5]              # 'llo'
l[:3]               # [2, 3, 5]; die ersten 3 Elemente
s[4:]               # 'o World!'; ab Index 4
s[::2]              # 'HloWrd'; jedes 2te Element
l[1::3]             # [3, 11]; ab Index 1 jedes 3te Element
```

Ein Slice erzeugt ein neues Objekt

Call-by-Object-Reference

C unterscheidet call-by-value und call-by-reference

Python ist keines von beiden, ähnlich Java

- Variablen sind nur Namen für Objekte
- → Mutability entscheidet, ob verändert oder neu erzeugt wird

```
### Mutable
def foo(bar):
    bar.append(42)
    print(bar) # [42]

a_list = []
foo(a_list)
print(a_list) # veraendert! [42]
```

```
### Immutable
def foo(bar):
    bar += 'new'
    print (bar) # 'oldnew'

a_list = 'old'
foo(a_list)
print(a_list) # unveraendert! 'old'
```

Ausgabe

Formatierte Ausgabe

print erlaubt beliebige Argumente

- Jedes übergebene Objekt wird mittels **str** druckbar gemacht
- Automatische Leerzeichen zwischen Argumenten

```
>>> print("Hello", 2.0, "World", 1)
Hello 2.0 World 1
```

Format-Strings

- Python unterstützt drei Varianten von Format Strings
- %-Operator für Strings; legacy Python 2.x
- **format**-Methode für Strings; ab Python 3.0
- f-Strings; beste Variante, ab Python 3.6

Placeholders siehe https://docs.python.org/3/library/string.html#formatspec

Formatierte Ausgabe: Beispiel

```
>>> a = 10; b = 12.3; c = "something"
```

%-Operator

```
>>> "a: %d b: %.2f c: %s" % (a, b, c)
'a: 10 b: 12.30 c: something'
```

format-Methode

```
>>> "a: {0} b: {1:.2f} c: {2}".format(a, b, c)
'a: 10 b: 12.30 c: something'
```

Empfohlene Variante: f-String

```
>>> f"a: {a} b: {b:.2f} c: {c}"
'a: 10 b: 12.30 c: something'
```