10.1.1 Einfache Datentypen in Python

Nichttyp

NoneType

Bemerkungen:

- Es gibt genau ein Objekt dieses Typs, nämlich None.
- Standardrückgabetyp und -wert von Funktionen

10.1.1 Einfache Datentypen in Python

Wahrheitstyp

bool	1byte	{false, true}

Beispiel:

b = true

10.1.1. Einfache Datentypen in Python

Operationen auf Wahrheitstyp

unäre Logik	not
binäre Logik	or and

Bemerkung:

- Python nutzt intensiv automatische Typumwandlung!
- None, numerische Nullen, leere Sequenzen und leere Abbildung werden als false angesehen.
- Alle anderen Werte werden als wahr interpretiert.

10.1.1 Einfache Datentypen in Python

Ganzzahliger Datentyp

int, long	jede Länge	l

Beispiele:

studenten = 250

grosseZahl = +2333003000000

Bemerkungen:

- Datenobjekte haben Datentyp. Variablen haben keinen Typ!
- Ältere Versionen unterscheiden int und long, wobei int dem long in C entspricht und long unbegrenzte Länge hat.

10.1.1. Einfache Datentypen in Python

Operationen auf Ganzzahlobjekten

unäre Arithmetik	+ -
binäre Arithmetik	+ - * / // % ** echte Division, Ganzzahldiv., Rest, Potenz
unäre Bitoperationen	~ (bitweises Komplement)
binäre Bitoperationen	<< >> ^ & | shift left, shift right, exor, und, oder
binäre Vergleiche	> >= == != <> <= < is is not Objektidentität, Nichtidentität
Zuweisung	= += -= *= /= //= %= **= &= |= ^= <<= >>=
unäre Operatoren	abs() int() long() float()
binäre Operatoren	divmod(,) (x/y , x%y)

10.1.1. Einfache Datentypen in Python

Gleitkommatypen

float	64 Bits	{ -1.7976...E+308 , ... , -4.9406...E-324 , 0 , 4.9406...E-324 , ... , 1.7976...E+308 }
Decimal	unbegrenzt	exakte Darstellung von Brüchen!

Beispiel:

f = 0.0

d = 5/3

10.1.1. Einfache Datentypen in Python

Operationen auf Fließkommaobjekten

unäre Arithmetik	+ -
binäre Arithmetik	+ - * / // % **
	echte Division, Ganzzahldiv., Rest, Potenz
binäre Vergleiche	> >= == != <> <= < is is not
	Objektidentität, Nichtidentität
Zuweisung	= += -= *= /= //= %= **=
unäre Operatoren	abs() int() long() float()
binäre Operatoren	divmod(,)
	(x/y , x%y)

10.1.1. Einfache Datentypen in Python

Komplexe Zahlen

complex 128 Bits (je 64 für Realteil und Imaginärteil im float-Format)

Beispiel:

c = 1.0 + 1.0j

10.1.1. Einfache Datentypen in Python

Operationen auf komplexen Zahlen

unäre Arithmetik	+ -
binäre Arithmetik	+ - * / ** echte Division, Potenz
binäre Vergleiche	< <= > >= == != <> is is not Objektidentität, Nichtidentität
Zuweisung	= += -= *= /= **=
unäre Operatoren	abs() .real .complex Realteil Imaginärteil

10.1.1. Einfache Datentypen in Python

Prioritäten bei Operatoren

14	, [ ] { } ' '	Paarbildung, Liste, Wörterbuch, Stringumwandlung
13	[ ] [:] . ( )	Indexierung, Teilbildung, Attributzugriff, Funktionsaufruf
12	+ - ~	unäre Operatoren
11	**	Potenzbildung
10	* / // %	Punktoperatoren
9	<< >>	Schiebeoperatoren
8	&	Bitweises Und
7	^	Bitweises exklusives Oder
6	|	oder
5	< <= > >= == != <> is is not in not in	Vergleiche, Identität, Mitgliedschaft
4	not	Logische Negation
3	and	Logisches Und

10.1.1. Einfache Datentypen in Python

Prioritäten bei Operatoren (Fortsetzung)

2	or	Logisches Oder
1	lambda	Anonyme Funktion

10.1.2. Datenstrukturen in Python

Zeichenketten

string	ASCII-Zeichenketten
unicode	UNICODE-Zeichenketten

Beispiele:

'Auto'

"Fahrzeug"

u'Fähre'

s = 'Ein neues Auto'

u = u"Ernährung"

10.1.2. Datenstrukturen in Python

Operationen auf Zeichenketten

< <= > >= == != <> is is not	binäre Vergleiche
x in s x not in s	Ist x Komponente von s? Ist x nicht...
s1 + s2	Aneinanderfügen von s1 und s2
s * n n * s	n Kopien von s (n ganze Zahl)
s[i]	i. Komponente von s, Beginn bei 0
s[i:j]	Teilstring von Index i bis (ausschl.) j
s[i:j:n]	Teilstring, aber nur jede n. Komponente
len(s), min(s), max(s)	Länge, kleinstes Zeichen, größtes Zeichen
reversed(s)	
sorted(s)	

10.1.2. Datenstrukturen in Python

Weitere Operationen auf Zeichenketten

s.capitalize()	Erstes Zeichen groß, alle anderen klein
s.lower()	
s.upper()	
s.center(width)	Zentriere s in neuem String vorgeg. Länge
s.count(sub, start, end)	Zähle Auftreten eines Teilstrings in Bereich
s.isalnum() s.isalpha() s.isdigit()	
s.islower() s.isupper()	
s.translate(table)	Kopie von s nach Abbildung mit Tabelle
...	

10.1.2. Datenstrukturen in Python

Listen

list

Beispiele:

[3, 4]

q = [2, 5]

p = [2, q, 6, [1,2]]

10.1.2. Datenstrukturen in Python

Operationen auf Listen

< <= > >= == != <> is is not	binäre Vergleiche
x in s x not in s	Ist x Komponente von s? Ist x nicht...
s1 + s2	Aneinanderfügen von s1 und s2
s * n n * s	n Kopien von s (n ganze Zahl)
s[i]	i. Komponente von s, Beginn bei 0
s[i:j]	Teilliste von Index i bis (ausschl.) j
s[i:j:n]	Teilliste, aber nur jede n. Komponente
len(s), min(s), max(s)	Länge, kleinstes Element, größtes Element
reversed(s)	
sorted(s)	

10.1.2. Datenstrukturen in Python

Weitere Operationen auf Listen

s[i] = x	i. Element auf x setzen
s[i:j:step] = t	Teilliste durch t ersetzen
del s[i:j:step]	Teilliste löschen
s.append(x)	Element anhängen
s.extend(x)	Liste anhängen
s.index(x,start,stop)	kleinster Index von x in Bereich
s.insert(i,x)	
s.remove(x)	
s.pop()	
s.reverse()	
s.sort()	

10.1.2. Datenstrukturen in Python

Wörterbücher

dict

- Wörterbuch ist eine Liste von Einträgen aus Schlüssel und Objekt.
- Über den Schlüssel kann auf das Objekt zugegriffen werden.
- Wörterbücher werden auch als assoziative Felder oder assoziative Speicher bezeichnet.

Beispiel:

telefonbuch = {'hans' : '4535', 'otto' : '4178', 'fritz' : '1624'}

telefonbuch['hans'] ergibt '4535'.

10.1.2. Datenstrukturen in Python

Operationen auf Wörterbüchern

< <= > >= == != <> is is not		binäre Vergleiche
len(d)		Anzahl der Einträge
dict()		leeres Wörterbuch
d[k]	d.get(k)	Objekt zu Schlüssel k
d[k] = x	d.setdefault(k,default)	Setze Objekt zu Schlüssel k auf x
del d[k]		Lösche Eintrag mit Schlüssel k
d.clear()		Lösche alle Einträge
k in d	d.has_key(k)	Ist der Schlüssel k in s vorhanden?
d.items()		Liste der Schlüssel-Objekt-Paare
d.keys()		Liste aller Schlüssel
d.values()		Liste aller Objekte
d.pop(k, default)		

10.1.2. Datenstrukturen in Python

Dateien

file	Datentyp für Dateien

Beispiel:

```
f = open('/tmp/beispiel','w')
f.write('Testbeispiel')
f.close()
```

10.1.2. Datenstrukturen in Python

Operationen auf Dateien

open(dateiname, zugriff)	Erzeuge Dateiobjekt mit Zugriff ('w' - write, 'r' - read, 'r+' - read and write, 'a' - append)
f.close()	Datei schließen
f.fileno()	Gib Dateinummer zurück
f.flush()	Leere Dateipuffer
f.read([size])	Lies Datei bis EOF oder size bytes
f.readline()	Lese Datei bis EOF in Liste von Zeilen
f.seek(offset)	Setze Lese-/Schreibkopf auf Position
f.tell()	Gib Lese-/Schreibkopfposition zurück
f.write(s)	Schreibe Zeichenkette in Datei
f.writelines(list)	Schreibe Liste von Zeichenketten in Datei

10.1.2. Datenstrukturen in Python

Mengen

set	änderbare Menge von eindeutigen Objekten
frozenset	feste Menge eindeutiger Objekte

Beispiel:

```
Elemente = ['Apfel', 'Birne', 'Orange', 'Pflaume']
Fruechte = set(elemente)
'Orange' in Fruechte
=> True
```

10.1.2. Datenstrukturen in Python

Operationen auf Mengen

len(s)		Anzahl der Elemente
e in s		true, wenn e in s enthalten ist, sonst false
for e in s		Iteriere über die Menge s
s1.issubset(s2)		Ist s1 Teilmenge von s2?
s1.issuperset(s2)		Ist s1 Obermenge von s2?
s.add(e)		
s.remove(e)		
s.clear()		Lösche alle Elemente
s1.intersection(s2)	s1 & s2	Schnittmenge zurückgeben
s1.difference(s2)	s1 – s2	
s1.symmetric_difference(s2)	s1 ^ s2	
s.copy()		

10.1.3. Ablaufsteuerung zwischen Anweisungen

Elementare Anweisungen

- Zuweisung – Namen mit Objekt verbinden
- Löschung – Namen von Objekt befreien
- Nichtanweisung (pass) – keine Aktion (etwa für Warteschleifen)
- Ausdruck – Berechnen eines Ausdrucks, etwa Funktion ohne Rückgabewert
- Ausgabeanweisung (print)
- Ladeanweisung (import) – Einladen und Initialisierung von Modulen
- Ausführung – Ausführen eines Programms
- Funktionsdefinition
- Klassendefinition

10.1.3. Ablaufsteuerung zwischen Anweisungen

Beispiele:

```
import sys
width = 20
del height
exec myProgramm.py
pass
```

Hinweis:

Alle Daten sind Objekte. Namen werden mit Objekten verbunden oder getrennt. Nicht mehr gebundene Objekte werden automatisch entfernt.

10.1.3. Ablaufsteuerung zwischen Anweisungen

Komposition - Blöcke

- Ein Block fasst mehrere Anweisungen zu einer Anweisung zusammen. Er wird durch Einrücken definiert!
- Blöcke können geschachtelt werden.

Beispiel:

...

```
    x = 0
    print 'x auf 0 gesetzt!'
    y = 2
    print 'y auf 2 gesetzt!'
```

...

10.1.3. Ablaufsteuerung zwischen Anweisungen

Alternation - if-Anweisung

'if' <AUSDRUCK>':'

<Anweisung>

{'elif' <Ausdruck>':'

<Anweisung>}

['else:'

<Anweisung>]

- Für <ANWEISUNG> kann eine elementare Anweisung, eine Komposition, eine Alternation oder eine Iteration stehen.
- Für <AUSDRUCK> muss ein Term vom Typ **bool** eingesetzt werden.

10.1.3. Ablaufsteuerung zwischen Anweisungen

Iteration - for-Anweisung

'for' <ELEMENT> 'in' <SEQUENZ>':'
    <ANWEISUNG1>

['else'
    <ANWEISUNG2>]

- <ELEMENT> ist ein Variablenname, der bei jedem Durchlauf mit der nächsten Komponente aus der <SEQUENZ> verbunden wird.
- <SEQUENZ> ist eine beliebige Liste, Menge oder Zeichenkette.
- <ANWEISUNG1> ist eine elementare Anweisung, Komposition, Alternation oder Iteration und wird mit jeder Belegung von <ELEMENT> ausgeführt.
- Nach dem Ende der Schleife wird <ANWEISUNG2> ausgeführt.

Beispiel:

```
for x in liste:
    print x
```

10.1.3. Ablaufsteuerung zwischen Anweisungen

Iteration - while-Anweisung

'while' <AUSDRUCK>':'

    <ANWEISUNG1>

['else:

    <ANWEISUNG2>]

- <AUSDRUCK> ist ein Ausdruck vom Typ **bool**. Falls true berechnet wird, wird <ANWEISUNG1> ausgeführt. Nach jedem Ausführen wird erneut getestet und ggf. <ANWEISUNG1> erneut ausgeführt.
- Wenn die Schleife beendet ist, wird <ANWEISUNG2> ausgeführt.

10.1.4. Unterprogramme in Python

Funktionen

Definition einer Funktion

'def' <NAME> '('<PARAMETERLISTE>'):'
    <ANWEISUNG>
    ['return' <RÜCKGABEWERT>]

Aufruf einer Funktion

<NAME>'('<PARAMETERLISTE>')'

Beispiel

```
def fib(n):
    " " " Fibonnaccireihe bis n ausdrucken " " "
    a,b = 0,1
    while b<n:
        print b,
        a , b = b , a+b
```

10.1.4. Unterprogramme in Python

Hinweise zu Funktionen

- Funktion muss keinen Wert zurückliefern.
- <PARAMETERLISTE> kann Standardwerte enthalten.
  def fib2(n = 10):
- Die <PARAMETERLISTE> kann beliebig lang sein.
  def myFunction(dateiname, *args):
- Man kann auch lambda-Formen wie in LISP verwenden.

```
def makeMultiplikator(factor):
    return lambda x: x * factor

MWST = makeMultiplikator(0.16)
MWST(10)
==> 1.6
```

10.1.5. Pakete in Python

Module

- Ein Modul ist eine Sammlung von Anweisungen, insbesondere von Funktionsdefinitionen, in einer Datei.
- Die Import-Anweisung ermöglicht die Nutzung eines Moduls in einem Programm bzw. anderen Modul.

  import meinModul.py
- Der Aufruf von Funktionen im Modul erfolgt in der Form

  <Modulname>'.'<Funktionsname>'('<Variablenliste>')'

10.1.5. Pakete in Python

Package

- Ein Paket (Package) ist eine Sammlung von Modulen, die in einem Unterverzeichnis zusammengefasst sind.
- Das Laden des ganzen Pakets erfolgt durch

  'from' <Paketname> 'import *'
- Das Laden einzelner Module erfolgt durch

  'from' <Paketname> 'import' <Modulname>
- Der Aufruf einzelner Funktionen erfolgt durch

  <Paketname>'.'<Modulname>'.'<Funktionsname>'('<Variablenliste>')'

10.1.6. OO-Konzepte in Python

Klassen

```
class <NAME>':'
        { <ANWEISUNG> |
          <KONSTRUKTORDEFINITION> }
```

- Die Definition von Komponenten erfolgt durch Zuweisung innerhalb der Klasse.
- Methoden werden als Funktionen innerhalb der Klassendefinition definiert, wobei 'self' als erstes Argument angegeben wird.
- Die Konstruktordefinition erfolgt durch Definition einer Methode mit dem Namen '__init__'.
- Der leere Konstruktor <NAME>() erlaubt die Instanziierung eines Objekts der Klasse.
- Die 'pass'- Anweisung erlaubt die Definition einer leeren Klasse, die selbst bzw. deren Instanzen erweitert werden können.

10.1.6. OO-Konzepte in Python

Klassen

Klassen sind in Python spezielle Datenobjekte!

Operationen:

- Referenzierung eines Attributs durch den Punktoperator
  (Ist sinnvoll, da Klassenattribut durch Setzen eines Wertes definiert wird.)
- Referenzierung einer Funktion durch den Punktoperator
  (Meist wird mit dem Klammeroperator direkt die Funktion aufgerufen.)
- Instanziierung zur Erzeugung eines Klassenobjekts
  (Notation mit Klammeroperator)

10.1.6. OO-Konzepte in Python

Beispiele zu Klassen

```
class Complex:
        def __init__(self, realpart, imagpart):
            self.r = realpart
            self.i = imagpart

x = Complex(1.0, 1.0)

x.r
    ==> 1.0


class Punkt:
        " " " Punkt in der Ebene " " "
        x = 3
        y = 2
        def print(self):
            print '(',self.x,',',self.y')'

p = Punkt()

p.print()

    ===> (2,3)
```

10.1.6. OO-Konzepte in Python

Vererbung

'class' <KLASSENNAME>'(' { <BASISKLASSENNAME>',' } '):'
{ <ANWEISUNG> |
<KONSTRUKTORDEFINITION> }

- Problem der Mehrfachvererbung gleichnamiger Attribute:
  Basisklassen von links nach rechts durchsuchen und
  stets zuerst die Basisklasse der Basisklasse und dann die Basisklasse selbst.
- Namen in der Klasse überschreiben stets Namen in den Basisklassen.

10.1.6. OO-Konzepte in Python

Bemerkungen

- Alle Attribute und Methoden sind öffentlich (public).
  Python verlässt sich auf Konventionen, um private oder geschützte Attribute oder Methoden zu ermöglichen.
- Trotz Tricks ist echtes Information Hiding nur mit Hilfe der Programmierung von Python-Funktionen in anderen Sprachen (etwa C++) möglich.
- Die Standardtypen sind auch Klassen im Sinne von Python, d.h. man kann von bool, long, string,... eigene Klassen ableiten!

10.1.7. Exceptions

Ausnahmen als Klassen

- Alle Ausnahmen sind Klassenobjekte.
- Der Interpreter fängt alle Ausnahmen,
  wenn sie nicht im Programm gefangen werden und
  behandelt sie durch Programmabbruch und Ausgabe einer Nachricht.
- Das Abfangen von Ausnahmen erfolgt mit try-Blöcken.

```
'try:'
        <Anweisung>

'except' <Ausnahmeklasse> ':'
        <Anweisung>

{ 'except' <Ausnahmeklasse> ':'
          <Anweisung> }

[ 'else:'
          <Anweisung> ]
```

10.1.7. Exceptions

Beispiele

```
(1) while True:
        try:
            x = int(raw_input("Bitte eine Zahl: "))
            break
        except ValueError:
            print "Es muss eine Zahl sein. Bitte nochmals versuchen!"


(2) import sys
   try:
        f  = open('datei.txt')
        s = f.readline()
        i  = int(s.strip())
   except IOError:
        print "IO Fehler"
   except ValueError:
        print "Keine Ganzzahl in Datei gefunden!"
```

10.2. Python und C++

Beispiel

```
// C++-Funktion
   char const* greet() {
         return „Hello, world“;
   }

// Boost.Python wrapper
   #include <boost/python.hpp>
   using namespace boost::python;
   BOOST_PYTHON_MODULE(hello) {
         def(„greet“, greet);
   }


" " " Python Interpreter " " "
   import hello
   print hello.greet()

===> Hello, world
```

10.2. Python und C++

Vorgehensweise

- C++-Code in hello.cpp in $(HOME)/CPP_PYTHON speichern
- Übersetzungsdatei erstellen:

```
#Hilfsdatei für Wrapper
subproject $(HOME)/CPP_PYTHON;
SEARCH on python.jam = $(BOOST_BUILD_PATH);
include python.jam;
extension hello                    # Deklaration einer Pythonerweiterung
: hello.cpp                        # Quelle
      <dll> $(BOOST_BUILD_PATH)/build/boost_python
      ;                            # Deklaration der Abhaengigkeit
```

- Übersetzungsdatei ausführen:

```
set PYTHON_ROOT    =  $(BOOST_BUILD_PATH)/bin
set PYTHON_VERSION =  2.2
bjam
```