Informatik
Hauptcampus
H O C H
S C H U L E
T R I E R

Einführung in PYTHON

Implementierung von Algorithmen: Datentypen, Kostenmodell, Test

Prof. Dr. Heinz Schmitz Stand 15. März 2017

Inhaltsverzeichnis

Allgemeine Informationsquellen

- Übersicht und Einordnung der Sprache
  http://de.wikipedia.org/wiki/Python_(Programmiersprache)
- PYTHON-Webseite http://python.org
- Dokumentationen http://docs.python.org/3/
  insbesondere mit empfehlenswertem Tutorial
- Standard-Bibliothek
  http://docs.python.org/3/library/index.html
- Pakete http://pypi.python.org/pypi
- Python-Style
  http://docs.python-guide.org/en/latest/writing/style/

Installationshinweise PYTHON

- Installieren Sie PYTHON auf Ihrem Rechner in der Version 3.x (aktuell 3.5)
  Download auf http://www.python.org/download/
- Hinweise zum Setup je Plattform (Unix, Windows, Mac) finden Sie auf
  http://docs.python.org/3/using/index.html

  In der Regel wird die Entwicklungsumgebung IDLE mit installiert und kann verwendet werden.
- PYTHON ist auch auf den Pool-Rechnern verfügbar
- Für das Schreiben von PYTHON-Programmen ist ein einfacher Editor ausreichend, eine Auswahl inkl. IDEs finden Sie unter

  http://wiki.python.org/moin/PythonEditors

  Verwenden Sie die Umgebungsvariable PYTHONPATH um Pfade zu Modulen anzugeben, die Sie importieren möchten ohne die Module in Ihr aktuelles Arbeitsverzeichnis kopieren zu müssen, z.B. für den Import von Modulen aus der Vorlesung.

Wir empfehlen allerdings die Verwendung von Eclipse mit dem PyDev-Plugin, siehe nächste Seite.

Installationshinweise Eclipse+PyDev

- Installieren Sie Eclipse http://www.eclipse.org/downloads/
- Installieren Sie das PyDev-Plugin, Installationshinweise finden Sie unter http://pydev.org/manual_101_root.html.
- Wechseln Sie zu *Window/Perspective/Open Perspective/Other*, dann *PyDev* wählen.
- Konfigurieren Sie den PYTHON-Interpreter für PyDev:
  - Gehen Sie zu *Einstellungen/PyDev/Interpreter - Python*.
  - Wählen Sie über *New* den Pfad zu Ihrer ausführbaren python-Datei, z.B. *python.exe* in Windows.
  - Wenn Sie nur eine PYTHON-Version installiert haben, bietet sich auch die Auto-Config-Funktion an.
- Richten Sie in Eclipse eine interaktive PYTHON-Konsole ein:
  - Gehen Sie zu *Window/ShowView/Console*.
  - Klicken Sie in dem erschienen Fenster rechts oben in der Ecke auf das Symbol für *New Console*.
  - Wählen Sie *PyDev Console* aus.
- Machen Sie sich mit den Verwendungsmöglichkeiten der PYTHON-Konsole in Eclipse vertraut, insbesondere mit dem Ausführen von markierten Code-Zeilen im Editor, siehe

  http://pydev.org/manual_adv_interactive_console.html

Import von Eclipse-Projekten der Vorlesung

Sie erhalten ggf. für die Lehrveranstaltung Dateien mit den Beispielen aus der Vorlesung, zum Teil auch Gerüste `stub_*` für das Lösen der Übungsaufgaben mit zugehörigen Testfällen.

- Importieren von *VL.zip*
  - Datei aus Stud.IP herunterladen.
  - Importieren über
    File/Import.../General/Existing Projects into Workspace
  - Auswahl *Select archive file* und heruntergeladenes Zip-Archiv suchen.
  - Stellen Sie sicher, dass das Projekt und *Copy projects into workspace* angehakt ist und bestätigen Sie mit *Finish*.
- Package-Struktur von *VL.zip* im Order `src`:
  - `vl.lek<nn>.*`: Beispiele der Vorlesung je Lektion
  - `ueb.<x>.*`: Vorlagen zur Lösung der Programmieraufgaben.
  - `test.<x>.*`: Unittests für Ihre Lösungen
- Entfernen Sie das Präfix `stub_` aus dem Dateinamen, wenn Sie eine Vorlage für das Lösen einer Übungsaufgabe verwenden. In den Testfällen ist der resultierende Dateiname und der richtige Pfad bereits hinterlegt.
- Starten Sie Ihre Progamme über *Run as .../Python Run* und die Testfälle mit *Run as .../Python unit-test*.

Verwendung

Wir verwenden PYTHON in verschiedenen Lehrveranstaltungen für die Implementierung von Algorithmen. Eine Zuordnung der Kapitel in diesem Dokument zu den Modulen

- Theoretische Informatik (Bachelor)
- Algorithmen-Design (Bachelor)
- Berechenbarkeit und Komplexität (Master)

zeigt die nachfolgende Tabelle. Außerdem wird eine Turing-vollständige Teilmenge von PYTHON als WHILE-Sprache verwendet. Die Kurzreferenz finden Sie im Anhang.

	Kapitel	ThI	AD	B+K
1	Interpreter, Funktionen, Schleifen	X	X	X
2	Zeichenketten	X	X	X
3	Mengen	X	X	X
4	Ganze Zahlen	X	X	X
5	Listen	X	X	X
6	Assoziative Listen	X	X	X
7	Tupel	X	X	X
8	Typen und Referenzen	X	X	X
9	Graphen		X	X
10	Stack und Queue		X	X
11	Generator Expressions		X	X
12	Set Comprehension		X	X
13	Iteratoren		X	X
14	Laufzeitmessung		X	X
15	Testfälle		X	X
A	WHILE-Programme	X		X

Literatur

[Het10] M. Hetland. *Python Algorithms*. Apress, 2010.

[Hä12] T. Häberlein. *Praktische Algorithmik mit Python*. Oldenbourg Verlag, 2012.

- ... ist eine universelle, interpretierte höhere Programmiersprache.
- ... hat eine interaktive PYTHON-Shell: direkte Auswertung von Ausdrücken und Ausführung von Anweisungen.
- ... wurde mit dem Ziel guter *Programmlesbarkeit* entworfen: möglichst einfacher und übersichtlicher Code durch wenige Schlüsselwörter und reduzierte Syntax.
- ... ist eine *Multiparadigmensprache*: strukturierte, objektorientierte, funktionale und aspektorientierte Programmierung werden unterstützt.
- ... hat eine dynamische Typverwaltung und *alles ist Objekt*! Der Typ einer Variable (Referenz auf ein Objekt) ergibt sich durch den Typ des Objekts.

1 Interpreter, Funktionen und Schleifen

Einführung in die PYTHON-Shell `py01_def.py`

- Ausdrücke, Anweisungen
- Verzweigungen, Schleifen
- Funktionsdeklarationen

2 Zeichenketten (`str`)

Einführung in PYTHON-Strings `py02_str.py`

- Zeichenketten, Konkatenation, Länge
- Indexzugriff
- Slicing

Eingabelänge

Sei $w = a_0 \cdots a_{m-1}$ für $m \geq 0$ eine Zeichenkette. Dann ist die *Länge von* w definiert als

$$|w| =_{\text{def}} m.$$

Kostenmodell

`w='abcde'`	$O(1)$
`w+v`	$O(\|w\| + \|v\|)$
`w*m`	$O(\|w\| \cdot m)$
`len``(w)`	$O(1)$
`w[i]`	$O(1)$
`w[i:j]`	$O(j - i)$
`a` **`in`** `w`	$O(\|w\|)$
`for` `a` **`in`** `w:`	$O(\|w\|)$

3 Mengen (`set`, `frozenset`)

Mengen in PYTHON `py03_set.py`

- Mengenoperationen und Vergleiche
- Unterschied zwischen **`set`** und **`frozenset`**

Eingabelänge

Für eine endliche Menge $A = \{e_0, e_1, \ldots, e_{m-1}\}$ ist die *Länge von* A definiert als

$$|A| =_{\text{def}} \sum_{i=0}^{m-1} |e_i|.$$

`len``(A)` liefert die Kardinalität $\#A$ von A, und nicht die Länge. Es gilt

$$\#A \leq |A|.$$

Kostenmodell

`A={1,2,3}`	$O(1)$
`len``(A)`	$O(1)$
`A|B`	$O(\#A + \#B)$
`A|=B`	$O(\#B)$
`A&B`	$O(\min(\#A, \#B))$
`A&=B`	$O(\#A)$
`A-B`	$O(\#A)$
`A-=B`	$O(\#B)$
`A<=B`	$O(\min(\#A, \#B))$
`A.remove(x)`	$O(1)$
`A.add(x)`	$O(1)$
`x` **`in`** `A`	$O(1)$
`for` `x` **`in`** `A:`	$O(\#A)$

4 Ganze Zahlen (`int`), Kurzschreibweisen

Zahlen aus $\mathbb{Z}$ in PYTHON `py04_int.py`

- arithmetische Operationen
- Kurzschreibweisen

Eingabelänge

Für $x \in \mathbb{Z}$ ist die *Länge von* x definiert als

$$|x| = |bin(abs(x))|.$$

Dabei ist $abs(x) = x$ für $x \geq 0$ und $abs(x) = -x$ sonst. Außerdem liefert bin : $\mathbb{Z} \mapsto \{0,1\}^+$ die Binärdarstellung einer positiven ganzen Zahl als Zeichenkette.

Kostenmodell

+, −, =, <,<=, >, >=	jeweils $O(1)$
==, !=	
and, **or**, **not**, **return**	jeweils $O(1)$
`a*b`	$O(\|a\| + \|b\|)$
ebenso für `//`, `%`	
`a**b`	$O(\|a\| \cdot \|b\|)$

Kurzschreibweisen

```
x = y = ... = a
x = a; y = b; ...
[x,y] = [a,b]
x += a auch für -, *, //, %
while b: s
for i in range(a,b): s
if b: s
```

5 Listen (`list`)

Listen in PYTHON `py05_list.py`

- Indexzugriff, Slicing
- Elemente ändern und löschen
- Listen aneinanderhängen, iterieren

Eingabelänge

Für eine Liste $l = [b_0, b_1, \ldots, b_{m-1}]$ ist die *Länge von* l definiert als

$$|l| =_{\text{def}} \sum_{i=0}^{m-1} |b_i|.$$

`len(l)` liefert die Anzahl Elemente m in l, und nicht die Länge. Es gilt

$$m \leq |l|.$$

Kostenmodell

`a=[1,2,3]`	$O(1)$
`len(a),a[i]`	$O(1)$
`a[i:j]`	$O(j-i)$
`a=b`	$O(1)$
`a+b`	$O(len(a) + len(b))$
`a+=b`	$O(len(b))$
`a==b`	$O(\min(len(a), len(b)))$
`del a[i]`	$O(len(a))$
`del a[-1]`	$O(1)$
`a.append(x)`	$O(1)$
`x in a`	$O(len(a))$
`for x in a:`	$O(len(a))$

6 Assoziative Listen (`dict`)

Mengen von Schlüssel-Wert-Paaren `py06_dict.py`

- Anlegen, Einfügen, Zugriff
- Elemente ändern und löschen
- iterieren

Eingabelänge

Sei $d = \{k_0 : v_0, k_1 : v_1, \ldots, k_{m-1} : v_{m-1}\}$ eine assoziative Liste. Die *Länge von* d ist definiert als

$$|d| =_{\text{def}} \sum_{i=0}^{m-1} |k_i| + |v_i|.$$

`len``(d)` liefert die Anzahl m der Schlüssel-Wert-Paare in d, und nicht die Länge. Es gilt

$$m \leq |d|.$$

Kostenmodell

`d={2:'B', 4:'D'}`	$O(1)$
`len``(d)`	$O(1)$
`d[k]`	$O(1)$
`del` `d[k]`	$O(len(d))$
`k` **`in`** `d`	$O(1)$
`for` `k` **`in`** `d:`	$O(len(d))$
`for` `k,v` **`in`** `d.items():`	$O(len(d))$

7 Tupel (`tuple`)

Tupel in PYTHON `py07_tuple.py`

- ähnlich wie **`list`**, aber unveränderbar

Eingabelänge

Für ein Tupel $t = (b_0, b_1, \ldots, b_{m-1})$ ist die *Länge von* t definiert als

$$|t| =_{\text{def}} \sum_{i=0}^{m-1} |b_i|.$$

`len``(t)` liefert die Anzahl m der Elemente in t und nicht die Länge. Es gilt

$$m \leq |t|.$$

Kostenmodell

`t=(1,2,3)`	$O(1)$
`len``(t)`	$O(1)$
`t[i]`	$O(1)$
`t[i:j]`	$O(j-i)$
`t+s`	$O(len(t) + len(s))$
`t+=s`	$O(len(s))$
`t==s`	$O(\min(len(t), len(s)))$
`x` **`in`** `t`	$O(len(t))$
`for` `x` **`in`** `t:`	$O(len(t))$

8 Typen und Referenzen

- Das Typkonzept von PYTHON ist *dynamisch*, d.h. jede Variable enthält immer nur eine *Referenz* auf ein Objekt. Zur Laufzeit kann diese Referenz geändert werden, unabhängig vom Typ des Objekts.
- PYTHON ist *streng getypt*: Jedes Objekt hat einen Typ und es werden keine impliziten Typumwandlungen vorgenommen (bis auf wenige Ausnahmen).
- Manche Objekte sind *unveränderbar* (immutable) und erlauben daher eine effiziente Verwaltung in Containern über ihren Hashcode. Vermeintliche Änderungsoperationen liefern neue Objekte.

`numbers`

`int`	repräsentiert $\mathbb{Z}$, Wertebereich unbeschränkt
`bool`	Werte `True`, `False`, Verhalten wie 0,1
`float`	
`complex`	

Sequences	endliche, geordnete Multimengen, Indizierung mit ganzen Zahlen, Slicing
`str`	immutable, daher hashable
`tuple`	immutable, daher hashable
`list`	mutable
...	

Set Types	endliche, ungeordnete Mengen, Elemente müssen immutable sein
`set`	mutable
`frozenset`	immutable, daher hashable

Mappings	endliche Mengen, Indizierung beliebig, Schlüssel immutable
`dict`	mutable

...

Objektreferenzen `py08_refs.py`

- Die Parameterübergabe bei Funktionsaufrufen erfolgt per *call by object reference*. Bei unveränderbaren Typen entspricht dies *call by value*, bei veränderbaren Typen *call by reference*.

9 Graphen

Liste von Adjazenzmengen

`py09_graph.py`
`py09_digraph.py`
`py09_edgecost.py`

- Graphen
- Digraphen
- Graphen mit Kantenkosten

Eingabelänge

Die Eingabelänge ergibt sich aus den verwendeten Datentypen. Für einen Graphen G mit m Knoten und k Kanten gilt stets

$$|G| \in O(m+k).$$

`len(G)` liefert die Anzahl der Knoten m und nicht die Länge von G. Es gilt

$$m, k \leq |G|.$$

Kostenmodell

`G` definieren	$O(m+k)$
`len(G)`	$O(1)$
`u in G[v]`	$O(1)$
`len(G[v])`	$O(1)$
`for u in G[v]:`	$O(deg(v))$ bzw. $O(outdeg(v))$
`G[u][v]`	$O(1)$

10 Stack und Queue

einfache Datenstrukturen `py10_stack.py`

- Stack (**`list`**)
- Queue (`deque` aus Modul `collections`)
- Min Priority Queue (Modul `heapq`)

Kostenmodell

`S.append(a)`	$O(1)$
`S[-1]`	$O(1)$
`S.pop()`	$O(1)$
`len``(S)`	$O(1)$
`Q.append(a)`	$O(1)$
`Q[0],q[-1]`	$O(1)$
`Q.popleft()`	$O(1)$
`len``(Q)`	$O(1)$
`heappush(P)`	$O(\log(len(P)))$
`P[0]`	$O(1)$
`heappop(P)`	$O(\log(len(P)))$
`len``(P)`	$O(1)$
`heapify(a)`	$O(len(a))$

11 Generator Expressions

iterierbare Ausdrücke `py11_genexpr.py`

- liefern einen Iterator zurück
- Erzeugung des nächsten Wertes bei Anforderung

$\Rightarrow$ Speicherplatzersparnis

Gute Lesbarkeit bei der Implementierung mathematischer Notationen, z.B.:

$\sum_{i=0}^{m-1} i^2$ `sum(i*i for i in range(m))`

$\sum_{\substack{i=0 \\ i \text{ ungerade}}}^{m-1} i^2$ `sum(i*i for i in range(m) if i%2)`

$\max_{0 \le i < m} s_i$ `max(s[i] for i in range(m))`

Kostenmodell

wie bei Ausprogrammierung

12 Set Comprehension

übersichtliche Mengendefinitionen `py12_setcomp.py`

- Schleifen und Bedingungen
- Potenzmenge
- auch für **`list`** , **`dict`**

Gute Lesbarkeit bei der Implementierung mathematischer Notationen, z.B.:

M=$\{0 \leq i < m \mid$ 3 teilt $i\}$ `M={i for i in range(m) if not i%3}`

N=$\{x^2 \mid x \in M\}$ `N={x*x for x in M}`

Kostenmodell

wie bei Ausprogrammierung

13 Iteratoren (`itertools`)

spezielle Iteratoren über kombinatorische Objekte `py13_iter.py`

- kartesisches Produkt: Potenzmenge und endliche Funktionen
- Permutationen: Anordnungsmöglichkeiten
- Kombinationen: k-elementige Teilmengen

Kostenmodell

`for t in product((0,1),repeat=m):`	$O(2^m)$
`for t in product(range(k),repeat=m):`	$O(k^m)$
`for p in permutations(range(m)):`	$O(m!)$
`for c in combinations(range(m),k):`	$O(\binom{m}{k})$

Wir vernächlässigen im Kostenmodell den Aufwand $O(m)$, der jeweils für die Erzeugung des nächsten Objekts entsteht, sondern zählen nur die Anzahl der Iterationen. Typischerweise liegt der Aufwand im Rumpf der Schleifen mindestens bei $O(m)$.

14 Laufzeitmessung (`timeit`)

`py14_timeit.py`

- Wiederholung von Messung und Aufrufen

15 Testfälle (`unittest`)

`py15_unittest.py`

- Testfälle mit mehreren Testmethoden
- Mehrere Testfälle bilden eine Testsuite

A WHILE-Kurzreferenz

Konstanten
$[-](0|(1|\ldots|9)(0|\ldots|9)^*)$

Bezeichner
$(a|\ldots|Z)(a|\ldots|Z|0|\ldots|9)^*$

Ausdrücke Konstanten und Variablen
`(a+b)`, `(a-b)`
`f(b1,...,bm)`

Bedingungen `(a==b)`, `(a!=b)`,
`(a>b)`, `(a>=b)`, `(a<=b)`,
`(a<b)`, `(not a)`, `(a or b)`,
`(a and b)`

Zuweisung `a = b`

Hintereinanderausführung
```
s1
s2
```

Bedingte Anweisung
```
if b:      if b:
     s1         s
else:
     s2
```

Schleifen
```
while b:
     s
for i in range(a1,a2):
     s
```

Funktionsdeklarationen
```
def f(a1,...am):
     s
     return c
def f(a1,...am):
     return c
```

Programme
```
f1
...
fm
```

Kostenmodell jeweils 1 Schritt:
`+`, `-`, `=`, `<`,`<=`, `>`, `>=`,`==`,`!=`,
`and`, **`or`**, **`not`**, **`return`**

For-Schleife wie While-Semantik

Eingabelänge ganzer Zahlen
$|x| = |bin(abs(x))|$