Programmeringsteknik I

Föreläsning 3a: Listor och strängar.

Johan Öfverstedt

Listor

Vi har sett i tidigare föreläsningar och laborationer att listor lagrar en serie objekt, som representeras av ett index:

xs: 0, 1, ..., len(xs) -> elem_0, elem_1, ..., elem_(len(xs)-1)

Index	0	1	2	3	4
Värde	'a'	'B'	'c'	'D'	'e'

Skivning (slicing) - manipulerande av dellistor

Skivning är en operation där man hämtar ut en delmängd av en lista på ett systematiskt sätt så att man kan läsa eller ändra den på plats.

```
xs = [1, 2, 3, 4, 5, 6, 7]
print(xs[1:-1:2])
>> [2, 4, 6]
print(xs[1:-2:2])
>> [2, 4]
xs[1::2] = [13, 15, 17]
print(xs)
>> [1, 13, 3, 15, 5, 17, 7]
```

Skivning kan användas för att hämta ut dellistor (med start:stop:steg notation)

och man kan tilldela till dellistan för att ändra den ursprungliga listan

Negativa tal för start/stopp-index betyder indexerat från slutet:
-n -> len(xs)-n

Skivning (slicing) - manipulerande av dellistor

Vi kan få en skivning som är i omvänd ordning genom att ange ett negativt steg, och ett `start` som är större än `stop`:

```
xs = [1, 2, 3, 4, 5, 6, 7]
print(xs[5:2:-1])
>> [6, 5, 4]
```

Skivningar, precis som range(start, stop, step),
använder alltid halvöppna interval:
[start, stop)
[2:5] -> index [2, 3, 4]

Skivning (slicing) - manipulerande av dellistor

Då -1 betyder sista elementet och -2 det näst sista elementet, osv (vilket underlättar då man slipper skriva len(xs)-1, len(xs)-2, ..., etc) kan man lätt råka göra misstag som döljs (då vi inte får ett felmeddelande)

```
xs = [1, 2, 3, 4, 5, 6, 7]
ys = []
for i in range(len(xs)-1):
    slice = xs[i-1:i+2]
    ys.append(sum(slice)/3.0)
print(ys)
>> [0.0, 2.0, 3.0, 4.0, 5.0, 6.0]
```

Logiskt fel. Baserat på summan av en tom skivning.

Skivning kan användas för att hämta ut dellistor (med start:stop:steg notation)

och man kan tilldela till dellistan för att ändra den ursprungliga listan

Skivning (adressering utanför)

Om man skivar en lista på ett sätt så att gränserna hamnar utanför så får man inget fel så som om man indexerar utanför:

```
x = [1, 2, 3]
x[5]
>> Traceback (most recent call last):
>>  File "<stdin>", line 1, in <module>
>> IndexError: list index out of range
x[4:5]
>> []
```

Listbyggande

Listor i Python kan byggas på många olika sätt.

- Uppräkning
- Konkatenering / sammansättning
- Upprepning av lista
- Tillägg via append och insert.
- Generator
- Listbyggare / list comprehension

Listbyggande: Uppräkning

Man kan skapa en lista genom att räkna upp en sekvens med värden.

```
xs = [1, 2, 3, 4, 5]
print(xs)
>> [1, 2, 3, 4, 5]
```

När man redan kan skriva ner alla element i listan (speciellt när de inte följer en enkel regel) är det enklast att räkna upp elementen.

I det givna exemplet hade det varit lättare att skriva:

```
xs = list(range(5))
```

Listbyggande: Konkatenering/Sammansättning

Givet två listor, kan man sätta samman dem till en längre lista:

```
xs1 = [1, 2, 3, 4, 5]
xs2 = [6, 7, 8, 9, 10]

xs3 = xs1 + xs2

print(xs3)
>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Listbyggande: Upprepning

Givet en lista, kan man upprepa den n antal gånger genom:

```
xs1 = [1, 2, 3, 4, 5]
xs2 = xs1 * 3

print(xs2)
>> [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Listbyggande: Tillägg med append/insert

Man kan bygga en lista, element för element med append eller insert.

`list.append(value)`: lägger till ett nytt element till listan sist.
`list.insert(index, value)`: lägger till ett nytt element på givet index

Listbyggande: Tillägg med append/insert

Man kan bygga en lista, element för element med append eller insert

```
xs1 = []
xs1.append(3)
xs1.append(4)
xs1.append(5)
xs1.insert(0, 1) # Lägger till 1 på index 0
xs1.insert(1, 2) # Lägger till 2 på index 1

print(xs1)
>> [1, 2, 3, 4, 5]
```

Listbyggande: Från generator

Man kan bygga en lista, genom att skriva en generator följt av en omvandling till en lista

```
def sq_gen(n):
    for i in range(1, n+1):
        yield i*i

xs = list(sq_gen(5))
print(xs)
>> [1, 4, 9, 16, 25]
```

Listbyggande: Listbyggare / List comprehension

Man kan bygga en lista på ett väldigt kompakt och smidigt sätt med listbyggar-syntaxen:

```
xs = [x for x in range(1, 6)]
print(xs)
>> [1, 2, 3, 4, 5]

xs = [x*x for x in range(1, 6)]
print(xs)
>> [1, 4, 9, 16, 25]
```

Listbyggande: Listbyggare / List comprehension

Man kan producera flera värden samtidigt genom att producera en tupel:

```
xs = [(x, x*x) for x in range(1, 6)]
print(xs)
>> [(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

eller genom att producera en lista:

```
xs = [[x, x*x] for x in range(1, 6)]
print(xs)
>> [[1, 1], [2, 4], [3, 9], [4, 16], [5, 25]]
```

Elementära algoritmer för listor

Find - linjärsökning - hittar index för ett eftersökt värde

```
def find(haystack, needle):
    for i in range(0, len(haystack)):
        if haystack[i] == needle:
            return i
    return None
```

Inbyggda funktioner i Python:
`haystack.index(needle)`

(ger ValueError om needle inte hittas)

Behöver i värsta fall `len(haystack)` iterationer och jämförelser (i de fall då det eftersökta värdet inte finns)

ÖVERKURS

Elementära algoritmer för listor

Find - binärsökning - hittar index för ett eftersökt värde i en sorterad lista

```
def find_sorted(haystack, needle):
    i1 = 0
    i2 = len(haystack)-1
    while i1 < i2:
        mid_ind = i1 + (i2-i1)//2
        if haystack[mid_ind] == needle:
            return mid_ind
        elif needle < haystack[mid_ind]:
            i2 = mid_ind
        else:
            i1 = mid_ind + 1
    return None
```

Behöver i värsta fall log_2(`len(haystack)`) iterationer och jämförelser (i de fall då det eftersökta värdet inte finns) [32 jämförelser för ~4 miljarder element i listan t.ex.]

log_2 är logarithm med bas 2

Elementära algoritmer för listor

Minimum (Maximum, byt < mot >)

```
def minimum(xs):
    assert(len(xs)>0)
    ind = 0
    value = xs[0]
    for i in range(1, len(xs)):
        if xs[i] < value:
            value = xs[i]
            ind = i
    return (value, ind)
```

Inbyggda funktioner i Python:
`min(xs)` ger minsta värdet,
`index(min(xs))` ger index för minsta värdet

Behöver alltid `len(xs)` iterationer och jämförelser då vi inte kan veta vilket det minsta eller största värdet är utan att undersöka alla.

Programmering med listor

Stora delar av programmering kan sammanfattas med:
Interaktion med hårdvaran: grafik, inmatning/utmatning från/till filer, nätverk, etc.

Listbyggande - Skapande av lista från existerande data eller från en algorithm.
Omvandling av listor (sortering, konvertering, aritmetik, etc) - från lista till lista.
Reducering av listor (minimum/maximum, summering, etc) - till ett sammanfattande värde.

Python är designat för att det ska vara enkelt att manipulera listor så att alla dessa steg är relativt lätta, och kräver väldigt lite kod.

Strängar

Strängar är sekvenser av tecken. Mycket av det vi kan göra med listor, kan vi även göra med strängar:

```
s = 'Hello, world!'
print(s)
>> Hello, world!
s_upper_list = [x.upper() for x in s]
print(s_upper_list)
>> ['H', 'E', 'L', 'L', 'O', ',', ' ', 'W', 'O', 'R', 'L', 'D', '!']
ss = ''.join(s_upper_list)
print(ss)
>> HELLO, WORLD!
```

`s.lower()` ger en sträng där alla versaler har omvandlats till gemener, och `s.upper()` gör det motsatta.

`s.join(lst)` sammansätter strängarna i `lst` med `s` mellan varje sträng.

Strängar

Strängar är sekvenser av tecken. Mycket av det vi kan göra med listor, kan vi även göra med strängar:

```
s = 'Hello, world!'
print(s[-2:1:-2])
>> drw,l
print(type(s[2]))
>> <class 'str'>
```


Om vi använder skivning och hämtar ett element i en sträng får vi alltså en ny sträng med ett tecken i, inte ett objekt av en tecken-typ. (som i t.ex. Java)

Strängar

Strängar är sekvenser av tecken. Mycket av det vi kan göra med listor, kan vi även göra med strängar:

```
s1 = 'abc'
s2 = 'abd'
print(s1 < s2)
>> True
print(s2 < s1)
>> False
```

Jämförelser av strängar sker lexikografiskt (tecken för tecken från vänster) [Detta gäller även för listor]

Strängar är oföränderliga

```
s = 'Hello, world!'
s[1:5]

>> 'ello'

>> s[1:5] = 'olle'

>> Traceback (most recent call last):
>>   File "<stdin>", line 1, in <module>
>>  TypeError: 'str' object does not support item assignment
```

Man kan inte ändra en sträng efter att den har skapats. Man kan bara skapa nya strängar med nytt innehåll.

Strängar är oföränderliga

```
s = 'Hello, world!'
ss = s[0:1] + s[4:0:-1] + s[5:]

print(ss)
>> Holle, world!
```

Finns under minilektion på kurshemsidan

Metod	Betydelse	Exempel

`count(string)`	*Räknar antalet (icke överlappande) förekomster av en sträng.*	`'axxxxbxx'.count('x') -> 6`
		`'axxxxbxx'.count('xx') -> 3`
		`'axxxxbxx'.count('xxx') -> 1`

`find(string)`	`s1.find(s2)` *returnerar index av den första förekomsten av* `s2` *i* `s1`*, eller -1 om* `s2` *inte förekommer i* `s1`*.*	`'abcd'.find('cd') -> 2` `'abcd'.find('cde') -> -1`
`format(v1, v2, ...)`	*Formatkonvertering. Se Formatering.*	`'{:.2f}'.format(3.1) -> '3.10'`
`index(string)`	*Som* `find` *men fel om strängen inte finns.*	
`isalpha()`	*True om bara bokstäver.*	`'ääöü'.isalpha() -> True`
`isdigit()`	*True om bara siffror.*	`'123'.isdigit() -> True` `'123.'.isdigit() -> False`
`join(string)`	*Förenar delarna. Se exempel.*	`'-'.join('abc') -> 'a-b-c'` `'**'.join('234') -> '2**3**4'`
`lower()`	*Ger en ny sträng där alla versaler är utbytta mot motsvarande gemener.*	`'Ab+Åd2'.lower() -> 'ab+åd2'`

`lstrip`
`rstrip`
`strip`
	Tar bort blanktecken i början,
	i slutet
	samt både i början och slutet.
		`' hej  '.lstrip()  ->  'hej  '`
		`' hej  '.rstrip()  ->  ' hej'`
		`' hej  '.strip()  ->  'hej'`

`partition(delim)`	*Returnerar en tuppel. Se exemplet.*	`'name@address'.partition('@') ->` `('name', '@', 'address')`
`replace(s1, s2)`	*Byter alla förekomster av* s_1 *mot* s_2*.*	`'Meraaker'.replace('aa','å') ->` `'Meråker'`

`split()`
`split(delim)`
	Gör en lista av delarna i strängen. Om parameter given används den som delare, annars används "white space".	`'ta det lugnt'.split() ->`
		`['ta', 'det', 'lugnt']`
		`'ta det lugnt'.split('t') ->`
		`['', 'a de', ' lugn', '']`

`upper`	*Ger en ny sträng där alla gemenerer är utbytta mot motsvarande versaler.*	`'ääö'.upper() -> 'ÄÄÖ'`