Compact Course Python

Michaela Regneri & Andreas Eisele

Lecture II

2010-04-07

Overview

- Functions
- Recursion
- Collection types:
  - Lists, Tuples
  - Sets
  - Dictionaries
- `for` loops
- `list` comprehensions

Functions

- Functions are reusable blocks of code belonging together
- When a function is called, its code is executed
- Functions have parameters (= arguments) they can access
- Functions can return values:
  In: `x = fun()`
  `x` is bound to the value
  returned by `fun()`
  via a `return` statement

```
def factorial (n):
    fac = 1
    i = n
    while i> 0:
        fac *= i
        i -= 1
    return fac
```

Function body

```
>>> factorial (4)
24
```


Syntax of Function Definitions

- Function definitions begin with the keyword `def`
- an arbitrary number (possibly 0) of parameters are separated by commas
- Return value is specified using a `return` statement; functions with no such statement or with an isolated return statement do not return any value

```
def fun (n, m, k):
    ....
    return ret
```

Function Calls

- Call with parameters p_1-p_n:
  function_name(p_1,p_2,...)
- Function calls are expressions that evaluate to the return value of the function
- When calling the function, parameter variables are instantiated with the values from the call (in the order listed):

```
def fun(n, m, k):
    print ('var', n, m, k)
    return m
```

```
> fun(1,2,3)
var 1 2 3
2
```

Return Value

Functions - Variables

- Functions can introduce or access *local* variables
  - the parameters
  - variables defined in the function
- Local variables are not visible outside the function
- Variables that are written to are assumed to be local; variables that are only read are assumed to be global
- *Manipulation* of variables within a method or function can use only local variables

```
def factorial(n):
    fak = 1
    i = n
    while(i > 0):
        fac *= i
        i -= 1
    return fak
```

```
counter = 0
def countup():
    counter + = 1
```

will not work!

Recursion

- Functions can call other functions
- In particular, functions can also call themselves; this is called *Recursion*
- In a recursive call, local variables can have different values on each incarnation of the function
- Recursion is a powerful tool which can be used to express many algorithms in an elegant way
- Caution: As with loops you have to pay attention to the fact that the recursion needs to end somewhere!

Recursion - factorial function

- the factorial function can be defined recursively:

```
def factorial (n):
    if n <= 1:
        return 1
    else:
        return n * factorial (n-1)
```

Base case for 0 and 1

Recursion with decreasing n

Recursion - Fibonacci

- The Fibonacci numbers is a sequence of numbers, defined recursively for natural numbers:
  fibonacci (0) = 0
  fibonacci (1) = 1
  fibonacci (n) = fib (n-1) + fib (n-2)

```
def fibonacci (n):
    if n <= 1:
        return n
    else:
        return fibonacci (n-1) + fibonacci (n-2)
```


Sequence types

- Sequence types are built-in data structures that combine multiple objects to one complex object
  - Lists: a collection of elements, fixed order, modifiable
  - Tuples: a collection of elements, fixed order, not modifiable
  - Sets: unordered collection of elements
  - Strings: sequence of characters (not modifiable)
  - Dictionaries: maps from keys to values
- for objects `s` from any sequence:
  - `len(s)`: Number of elements in s
  - `s.clear()`: Removes all elements from (modifiable) s
  - `s1 == s2`: (Value) equality of `s1` and `s2`

Lists

- A list is an ordered collection of values
- You can write it as literal:
  `list = ['a', 'Hello', 1, 3.0,[1,2,3]]`
- the list items do not have to have the same type (so)
- Access to list items with indices:

```
> list[0]
'a'
```

```
> list[-1]
[1,2,3]
```

```
> list[-1][1]
2
```

```
> list[5]
IndexError: list index out of range
```


Lists - methods and operators (1)

- Add items:
  - append an element: `list.append(elem)`
  - insert element at position i: `list.insert(i,elem)`
- Concatenating lists:
  - either: `newlist = list1 + list2`
  - or: `list1.extend(list2)`
- Delete elements:
  - `li.remove(el)` deletes the first `el` in the list `li`
  - `del li[n]` deletes the element with index n
- Membership and non-membership (slow for long lists):
  `elem in list` or `elem not in list`

Lists - methods and operators (2)

- Index of the first occurrence of elem in list: `list.index(elem)`
- How often is elem in list? `list.count(elem)`
- Reverse a list: `list.reverse()`
- Sort: `list.sort ()` (Only with same type)

```
> Li = [5,2,7]
> Li.reverse ()
> Li
[7, 2, 5]
```

```
> Li = [5,2,7]
> Li.sort()
> Li
[2, 5, 7]
```

```
> Li = [[1,2],[1,2,3],[3,2],[1,3]]
> Li.sort()
> Li
[[1, 2],[1, 2, 3],[1, 3],[3, 2]]
```


Lists - methods and operators (3)

- lists can be “multiplied” by integer numbers:

```
> Li = [1,2,3]
> Li = Li * 3
> Li
[1,2,3,1,2,3,1,2,3]
```

- `list * n` specifies a list containing n repetitions of the contents of `list`; `n <= 0` is the empty list

- Warning: this will not generate so-called *deep* copies of the list! (More on this later)

```
> Li = [[]] * 3
> Li[0]. append(1)
> Li
[[1],[1],[1]]
```

lists - *slicing* (1)

- the slicing operator can return a part of a given list
  - `list[i:]` is the partial list of i to the end of `list`
  - `list[i:j]` is the partial list of `i` to (but excluding) `j`
  - `list[i:j:k]` makes steps of size `k`

```
> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
> numbers[2:8]
[2, 3, 4, 5, 6, 7]
> numbers[2:8:2]
[2, 4, 6],
> numbers[8:2: -1]
[8, 7, 6, 5, 4, 3]
> numbers[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```


Lists - *slicing* (2)

- Using slicing, lists can be modified in an elegant way

`del list[0:3]` deletes the first 3 elements in `list`

`del list[0:5:2]` deletes every second entry from first to 5th element in `list`

`list1[0:3] = list2` replace the first 3 elements of `list1` by the elements of `list2`

`list1[0:5:2] = list2` replace every second entry from 1 and up to the 5th element in `list` by successive entries of `list2` (list2 must contain as many elements as `list1[0:5:2]`!)

Lists - *slicing* (3)

```
> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> numbers[2:5] = [2,2,3,3,4,4]
> numbers
[0, 1, 2, 2, 3, 3, 4, 4, 5, 6, 7, 8, 9, 10]
> numbers[0:9:2] = ['a', 'a', 'a', 'a', 'a']
> numbers
['a', 1, 'a', 2, 'a', 3, 'a', 4, 'a', 6, 7, 8, 9, 10]
```


Tuple: `tuple`

- similar to lists: `('a', 1, 'b')` but not modifiable
- Initializing:
  - 0 items: `tuple = ()`
  - 1 item: `tuple = elem,`
  - more items: `tuple = elem1, elem2, elem3`
- access to elements with [] and slices
- more efficient than lists
- *sequence unpacking:* (Also works well with lists)

```
> t = 'a', 2,[2,3]
> x, y, z = t
> z
[2,3]
```

Sets

- Sets are unordered collections of items that cannot contain any duplicate element

```
> numbers = [1, 2, 3, 1, 1]
> nSet = set(numbers)
> nSet
{1, 2, 3}
```

- as a literal: `nSet = {1,2,4,5}` (Empty set: `set()`)
- or defined indirectly via a different sequence type: `set(myList)`
- duplicate items are eliminated
- efficient test of values for set membership (much faster than lists!)
- *sets may only contain immutable types!* (Numbers, strings, tuples of immutable values, booleans, ...)

Sets - methods and operators (1)

- `Add elem`: `mySet.add(elem)`
- `Remove elem`
  - `set.remove(elem)` (Error if `elem` not available)
  - `set.discard(elem)` (Removed `elem` if available)
- Add all elements from `set2` to `set1`:
  `set1.update(set2)`
- Membership and non-membership:
  `elem in set` or `elem not in set`

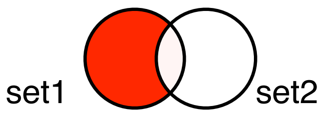
Sets - methods and operators (2)

Methods can have other aggregate types as 2nd argument; operators require two sets.

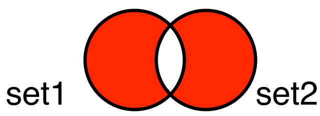
- Subset / superset (Return: True / False):
  - `set1.issubset(set2)` or `set1.issuperset(set2)`
  - `set1 <= set2` or `set1 >= set2`
- Union / intersection
  (Returns: the new set)
  - `set1.union(set2)` or `set1.intersection(set2)`
  - `set1 | set2` or `set1 & set2`

Sets - methods and operators (3)

- Difference set (return: new set with elements from `set1` but not in `set2`)
  - `set1.difference(set2)`
  - `set1 - set2`



- Symmetric set difference (return: new set with elements that are either in `set1` or `set2`, But not both)
  - `set1.symmetric_difference(set2)`
  - `set1 ^ set2`



Sets - methods and operators (4)

- all set operations are also available as 'update' method / operator
- no return value, `set1` will obtain the resulting set:
  - `set1.difference_update(set2)`
    `set1 -= set2`
  - `set1.symmetric_difference_update(set2)`
    `set1 ^= set2`
  - `set1.intersection_update(set2)`
    `set1 &= set2`
  - `set1 |= set2`

Invariant sets:
frozenset

- there is a constant set variation, the `frozenset`
- works like set: `fs = frozenset(collection)`
- But all the methods that add elements, delete, modify or are forbidden (`add`, `remove`, `discard`, all `update` methods)
- All other methods and operators work in `set` (and give back `frozenset` instead `set`)
- Motivation: frozensets can be used in places where only immutable values are allowed, e.g. as members of other sets or keys of dictionaries

Initialization of lists, sets, etc.

- the collection types which are not ditionaries can directly convert into each other
- Achieved via `typename(collection_instance)` - See sets

```
> mySet = set([1,2])
> myList = list(mySet)
> myTuple = tuple(mySet)
> tuple2 = tuple(myList)
> set2 = set(myList * 5)
...
```


Dictionaries: `dict`
(aka maps, hashes, associative arrays)

- Dictionaries are mappings from (unique) keys to values; the key must have an immutable type
- Access to values via the key
- For example, a phone book:

```
> Tel = {'Mueller': 7234, 'Meier': 8093}
> Tel['Meier']
8093
> Tel['Smith'] = 2104
> Tel
{'Mueller': 7234, 'Meier': 8093, 'Smith': 2104}
```

Dictionaries as literals

- `{}` is an empty dictionary (!), the same as `dict()`
  ==> hence `{}` cannot be used for an empty set
- some alternative spellings with the same result:

```
> Tel = {'Mueller': 7234, 'Meier': 8093}
```

```
> Tel = dict([['Mueller', 7234],['Meier', 8093]])
```

```
> Tel = dict([('Mueller', 7234), ('Meier', 8093)])
```

only with a valid variable name as key

```
> Tel = dict(Mueller = 7234, Meier = 8093)
```


Dictionaries - keys (1)

- Keys must have constant values (see sets)
- `frozenset` is therefore permitted (as in sets)

```
> Tel = {}
> Tel[['Peter', 'Sophie']] = 7473
Traceback (most recent call last):
  File <stdin> '', Line 1, in <module>
TypeError: unhashable type
```

```
> Tel[frozenset(['Peter', 'Sophie'])] = 7473
> Tel
{frozenset (['Peter', 'Sophie']): 7473}
```

Dictionaries - keys (2)

- keys for which the comparison with "==" gives `True` are considered equal
- if a key is used that it is already in the dictionary, it obtains the new value, the old one is deleted

```
> Tel['Peter'] = 7473
> Tel['Peter'] = 9999
> Tel
{'Peter': 9999}
```

- Caution: `1` and `1.0` are therefore the same key!

Dictionaries - methods (1)

- Test whether a key `key` exists in `dict`:
  - `key in dict`
- Deleting a key / value pair (`key: value`):
  - `del dict[key]` (Returns nothing)
  - `dict.pop(key)` (returns `value`)
- Setting the key `key` to the value `value`, if `key` does not exist:
  `dict.setdefault(key, value)`
  (If `key` exists, the old value of `key` is returned, otherwise `value`)

Dictionaries - methods (2)

- complement `dict1` with keys/values from `dict2`
  `dict1.update(dict2)`
  (Keys that are in both, get the value from `dict2`)
- "View" of all key: `dict.keys ()`
- "View" of all values: `dict.values ()`
- "View" of all key-value pairs: `dict.items ()`

Caution: the order is not deterministic! The only guarantee: two calls in succession on the same system without any change of `dict` deliver the same sequence, corresponding to keys and values.

Dictionaries - "views"

- Views look like this:

```
>>> map
('A': 1, 'l': 3, 'o': 4)
>>> map.keys()
dict_keys(['a', 'l', 'o'])
```

- they both reflect the current state of the dictionary
- we regard them as a collection types that we cannot change (not as *immutable*)
- further manipulation is possible after conversion to list:
  `list = list (map.keys ())`

Lists – tuples – sets – when to use what?

- fixed order, with methods to change elements: `list`
- fixed order, no methods or manipulation (fixed): `tuple`
- no particular order, manipulated: `set` (Much more efficient for membership testing compared to lists)
- invariant sets: `frozenset`
- immutable types as keys (in `dict`) and elements of sets (in `set` and `frozenset`)

`for`

- `s` is a collection type
- iterates over every element in `s`
- `i` is the element currently considered
- at each iteration `block` is executed
- `break`, `continue` and `else` function as for `while`

```
for i in s:
  block
```

```
> list = [1, 'a', True]
> for i in list:
.. print (i)
..
1
a
True
```

for

- Average of all list items using `for`:

```
def average (list):
    result = 0.0
    for number in list:
        result += number
    return results / len(list)
```

- Alphabetical key-value pairs:

```
def sortedprint(map)
  key = sorted(map.keys())
  for key in key:
      print(str(key) + ':' + str(map[key]))
```

`sorted (map)` and `sorted (map.keys ())` are equivalent

for with dictionaries

- `for` can iterate also over the "View" objects of dictionaries (`keys`,`items`,`values`)
- you often want to iterate over all pairs in a dictionary, thanks to "sequence unpacking" it simply goes like this:

```
def oneLinePerEntry(map):
    for key, val in map.items():
        print (str(key) + ':' + str(val))
```

Function/method definitions – some advanced features

- Functions can have optional arguments, arbitrary numbers of arguments, and arguments specified via keywords
- The exact functionality may depend on the function call:
  - `max(a₁,a₂,...,aₙ)` --> return maximum of n arguments
  - `max(sequence)` --> return maximal element of one argument
- Optional arguments are specified by giving a default value in the function definition (Value is shared between calls !!!)
- Arbitrary numbers of arguments are matched against a (tuple) parameter preceded by an asterisk in the definition
- Arbitrary keyword arguments are matched against a (dict) parameter preceded by a double asterisk in the definition

Function/method definitions – some advanced features

```
def test(a,b,c=33,d=44,*e,**f): print (a,b,c,d,e,f)

>>>test(1,2)
1 2 33 44 () {}
>>>test(1,2,3,4,5,6)
1 2 3 4 (5, 6) {}
>>>test(k1=1,k2=2,b=3,a=4)
4 3 33 44 () {'k2': 2, 'k1': 1}
>>>test(999)
TypeError: test() takes at least 2 positional arguments
(1 given)
```

Building sequences (1): `range`

- The type `range` is used to create sequences of consecutive numbers
- `range` does not (longer) return a list, but an *iterator*-like collection type
- Iterators can be used with for loops
  - `range(m)` corresponds to the elements `[0,1 ,..., m-1]`
  - `range(n, m)` ≈ `[n, n+1, ..., m-1]`
  - `range(n, m, k)` does steps of size k (as in slicing)

Building sequences (2): `enumerate`

- Sometimes, we want to iterate over sequence elements and indices at the same time, e.g. in order to
  - remember the location of certain elements
  - check constraints between neighbouring elements
  - compute statistics over the location of elements
- `enumerate(sequence)` returns pairs `(index,value)` where `index` is from `range(0,len(sequence))`

```
for i, val in enumerate(seq):
    do_something(i,val)
```

<~>

```
for i in range(len(seq)):
    do_something(i,seq[i])
```

- `enumerate` is more general, also works with sequences that can be traversed only once (e.g. while reading a file)

Building sequences (3): `zip`

- Sometimes, we want to iterate over several sequences in parallel and generate tuples
- `zip(seq1,seq2,...,seqN)` iterates over n-tuples of corresponding values `(val1,val2,...,valN)` where `val_i` is taken from `seq_i`
- A snippet from http://norvig.com/python-iaq.html :

  Q: Hey, can you write code to transpose a matrix in 0.007KB or less?
  A: I thought you'd never ask. If you represent a matrix as a sequence of sequences, then zip can do the job:
  >>> m = [(1,2,3), (4,5,6)]
  >>> zip(*m)
  [(1, 4), (2, 5), (3, 6)]
  To understand this, you need to know that `f(*m)` is like `apply(f,m)...`

Dictionaries with default values – `collections.defaultdict`

- `defaultdicts` are very convenient for counting/collecting events found in streams of data
- You need to specify the type of the default values
- Useful options include: `int, list, set,` as well as embedded `defaultdicts`

```
>>> from collections import defaultdict
>>> di = defaultdict(int)
>>> for c in “Hello”: di[c] += 1
>>> di
defaultdict(<class 'int'>, {'H': 1, 'e': 1, 'l': 2,
'o': 1})
>>> di[‘a’]
0
```

List comprehensions

- Very compact, yet readable way to generate lists from simpler list, inspired by the set builder notation in mathematics and similar constructs e.g. in Haskell
- General form:
  [*expression for_loop*$_1$ ... *for_loop*$_n$ *if_clause*$_1$ ... *if_clause*$_k$]
- Often used to create auxiliary representations for sorting, extracting interesting cases etc.
- Can be nested to build nested lists
  (at the cost of reduced readability!)

List comprehensions – examples

- Build a table of powers of small integers:
  ```
  [[i**n for n in range(1,5)] for i in range(11)]
  ```
- Build strings with certain properties:
  ```
  s = [‘’]
  for i in range(5):
     s = [x+c for x in s for x in ‘abc’]
  s = [x for x in s if ‘aba’ in x]
  ```
- Find key with largest value in a dict:
  ```
  _,key = max([(val,key) for key,val in d.items()])
  ```

Summary

- Functions
- Recursion
- Collection types: lists, tuples, sets, dictionaries
- New control structure: for loop
- List comprehensions