Introduction to Programming

4. Data Structures

Mgr. Tomáš Kuzma

Center for Cognitive Science
Department of Applied Informatics
Faculty of mathematics, physics and informatics
Comenius University in Bratislava

October 12th, 2016

Sequences

Some useful built-ins

- add numbers to elements – `enumerate(iterable)`:

```
>>> x = ['a', 'b', 'c', 'd']
>>> enumerate(x)
<enumerate object at 0x7f602484e750>
>>> list(enumerate(x))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

- the `()` delimited sequences are `tuples`, basically unmodifiable lists

- assemble *n*-tuples from *n* iterables – `zip(iterables...)`:

```
>>> first = ['Adam', 'Bob', 'Chuck']
>>> last  = ['Green', 'Black', 'Gray']
>>> names = zip(first, last)
>>> names
<zip object at 0x7efc31bebc48>
>>> list(names)
[('Adam', 'Green'), ('Bob', 'Black'), ('Chuck', 'Gray')]
```

Multiple assignment

- python supports multiple (simultaneous) assignment:

```
>>> a, b = 3, 4
>>> a
3
>>> b
4
```

- it works on any iterable:

```
>>> x = [+4, -2]
>>> low, high = sorted(x)
>>> low
-2
>>> high
4
```

Multiple assignment

- it also works in `for` cycles:

```
>>> l = [[2, 2], [3, 5], [4, 6]]
>>> for a, b in l:
...     print(str(a)+' + '+str(b)+' = '+str(a+b))
...
2 + 2 = 4
3 + 5 = 8
4 + 6 = 10
```

- and list comprehensions:

```
>>> first = ['Adam', 'Bob', 'Chuck']
>>> last  = ['Green', 'Black', 'Gray']
>>> names = [f+' '+l for f, l in zip(first, last)]
>>> names
['Adam Green', 'Bob Black', 'Chuck Gray']
```

List unpacking: tail

- the number of sources and targets must match:

```
>>> a, b = [1, 2, 3, 4, 5]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

- but, we can mark *one* output with * to collect the rest:

```
>>> a, *b = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4, 5]
```

List unpacking: head

- the collection argument may collect nothing:

```
>>> a, b, *c = [1, 2]
>>> a
1
>>> b
2
>>> c
[]
```

- the collecting argument can at any place

```
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
```

List unpacking: iterables

- list unpacking works on any iterable:

```
>>> initial, *rest = 'Tommy'
>>> initial
'T'
>>> rest
['o', 'm', 'm', 'y']
```

- we can also unpack into function arguments:

```
>>> x = [7, 2]
>>> a, b = divmod(*x)
>>>
>>> a
3
>>> b
1
```

Unnecessary output

- it is *convention* to mark ignored results as _:

```
>>> x = [5, 2, 8, 3, 9, 6]
>>> low, *_, high = sorted(x)
>>> low
2
>>> high
9
>>> for _ in range(5):
...     print('!')
...
!
!
!
!
!
```

  - i.e. do this 5 times, index is not used

Sets

Set creation

- `class set` – unordered bag of *distinct* elements
- creating an empty set:

```
>>> x = set()
>>> x
set()
```

- set *literal*:

```
>>> {4, 4, 1, 5}
{1, 4, 5}
```

  - caveat: `{}` is *not* an empty set, use `set()`

- creating from any iterable:

```
>>> set([3, 2, 5, 2, 1, 3, 6])
{1, 2, 3, 5, 6}
```

  - don't depend on the order (implementation specific)

Viewing sets

- fast `in` operator – sets:

```
>>> x = {1, 2, 3}
>>> 2 in x
True
>>> 4 in x
False
>>> 'abc' in x
False
```

- sets are iterable:

```
>>> x = {4, 3, 2, 3, 4}
>>> for i in x:
...     print(i)
...
2
3
4
```

- slow `in` operator – lists, strings, …:

```
>>> s = 'bafta'
>>> 'a' in s
True
>>> 'c' in s
False
>>> 50 in range(1):
False
```

- `len` function returns number of items:

```
>>> x = {1, 2, 3, 4}
>>> len(x)
5
>>> x = {4, 3, 2, 3, 4}
>>> len(x)
3
```

Modifying sets

`add(x)`	add new value to set (nothing if already a member)
`remove(x)`	remove value from set (*error* if not a member)
`discard(x)`	remove value from set (*ignore* if not a member)
`clear()`	remove all elements from set
`pop()`	remove one arbitrary element from set returns the element (*error* if set is empty)

Set operators

- relational operators:

math	operator	function
$\subset$	`a < b`	—
$\subseteq$	`a <= b`	`a.issubset(b)`
$\supseteq$	`a >= b`	`a.issuperset(b)`
$\supset$	`a > b`	—

- binary operators:

math	operator	function	
$\cup$	`a | b`	`a.union(b)`	in a or in b (or both)
$\cap$	`a & b`	`a.intersection(b)`	in bot a and b
$-$	`a - b`	`a.difference(b)`	in a, but not in b
Δ	`a ^ b`	`a.symmetric_difference(b)`	in only one of a or b

- compound assignments:

operator	function
`a |= b`	`a.update(b)`
`a &= b`	`a.intersection_update(b)`
`a -= b`	`a.difference_update(b)`
`a ^= b`	`a.symmetric_difference_update(b)`

Set comprehensions

- *set comprehension* – very similar to list comprehension:

```
>>> t = 'TO BE Or not to be'
>>> s = {x.lower() for x in t.split()}
>>> s
{'be', 'not', 'or', 'to'}
>>> l = ['2A', '4B', '4A', '5B', '6C']
>>> r = {x[0] for x in l if x[1] != 'C'}
>>> r
{'4', '2', '5'}
```

- `{...}` does the same as `set([...])`, but more efficiently

Mappings

Mapping types

- *sequence types* – an ordered sequence of items
- *mapping types* – mapping *keys* to *values*:
  - keys are unique
  - values don't have to be
  - one value for each key contained
- basic sequence type – list
- basic mapping type – dictionary (`dict`)

Dictionary: `dict`

- `class dict`: map of keys to values
- creating an empty dictionary:

```
>>> d = dict()
>>> d
{}
>>> e = {}
>>> e
{}
```

  - remember: `{}` is not an empty set

- dictionary literal:

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d
{'two': 2, 'one': 1, 'three': 3}
```

dict access

- dict item can be also accessed with []:

```
>>> d = {'abc': 5, 'def': 3}
>>> d['abc']
5
>>> d['wtf']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'wtf'
```

- the same goes for writing:

```
>>> d['def'] = 4
>>> d['ghi'] = 6
>>> d
{'abc': 5, 'def': 4, 'ghi': 6}
```

  - note that the 'ghi' key was added by the assignment

dict access

- the `len(...)` built-in returns the number of keys:

```
>>> d = {2: [3, 4, 5], 'b': 'c'}
>>> len(d)
2
```

- the `in` operator works on `dict`s (and it's also fast):

```
>>> d = {'abc': 5, 'def': 3}
>>> 'abc' in d
True
>>> 'wtf' in d
False
```

- the `del` operator key removes a key (and its value):

```
>>> del d['abc']
>>> d
{'def': 3}
```

dict access

- the `dict.get(key)` method also returns value for key:

```
>>> d = {'abc': 5, 'def': 3}
>>> d.get('abc')
5
>>> d.get('wtf')
>>>
```

  - what? no error? nothing?

```
>>> x = d.get('wtf')
>>> print(x)
None
>>> type(x)
<class 'NoneType'>
```

    - hmm...

Iterating

- `dict` itself is iterable (over *keys*):

```
>>> d = {'abc': 5, 'def': 3, 'ghi': 6}
>>> for x in d:
...     print(x)
...
ghi
abc
def
```

- we can also iterate also `.values()`:

```
>>> d = {'abc': 5, 'def': 3, 'ghi': 6}
>>> for x in d.values():
...     print(x)
...
6
5
3
```

Iterating

- finally, we can iterate through both keys and values at once:

```
>>> d = {'abc': 5, 'def': 3, 'ghi': 6}
>>> for k, v in d.items():
...     print(k + ': ' + str(v))
...
ghi: 6
abc: 5
def: 3
```

- `dict.items()` actually yields `(key, value)` tuples:

```
>>> for x in d.items():
...     print(x)
...
('def', 3)
('ghi', 6)
('abc', 5)
```

Other initializations

- *dictionary comprehensions* work like expected:

```
>>> d = {i: i**2 for i in range(1,7)}
>>> d
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

- construction from *key* and *value* iterables:

```
>>> keys = ['abc', 'def', 'ghi']
>>> values = ['ABC', 'DEF', 'GHI']
>>> dict(zip(keys, values))
{'ghi': 'GHI', 'abc': 'ABC', 'def': 'DEF'}
```

- construction from a `(key, value)` iterable:

```
>>> dict(enumerate(['ab', 'cd', 'ef']))
{0: 'ab', 1: 'cd', 2: 'ef'}
```

`defaultdict` utility class

- `defaultdict` class from the `collections` module:

```
>>> import collections
>>> d = defaultdict(int)
>>> d[4]
0
>>> d[5] += 3
>>> d
defaultdict(<class 'int'>, {4: 0, 5: 3})
```

- reading a non-existing key yields a *default* value
  - create by `defaultdict(type)`, type decides* value:
    - `int` -> `0`
    - `float` -> `0.0`
    - `bool` -> `False`
    - `str` -> `''`
    - `list` -> `[]`
    - `set` -> `set()`
    - `dict` -> `{}`