NPRG065: Programming in Python

Lecture 3

Tomas Bures
Petr Hnetynka
{(bures,hnetynka)@d3s.mff.cuni.cz}
Lists

• Dynamic arrays
  ▪ mutable

```python
squares = [1, 4, 9, 12, 25]
squares[3] = 16
print(squares)  # -> [1, 4, 9, 16, 25]
```

• Indexing and slicing like with strings

```python
squares[-1]  # -> 25
squares[-3:]  # -> [9, 16, 25]
```

▪ warning: slicing returns a new list

```python
squares[:]  # -> [1, 4, 9, 16, 25]
# a copy of the whole list
```
Lists

• Concatenation via +
  ▪ returns a new list

```
squares + [36, 49, 64, 81, 100]    # ->
    # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

• append() method
  ▪ adding at the end of the list
    • modifying the list

```
squares.append(36)
print(squares)    # -> [1, 4, 9, 16, 25, 36]
```
Lists

- Assignment to slices

```python
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
    # -> ['a', 'b', 'C', 'D', 'E', 'f', 'g']
letters[2:5] = []  # -> ['a', 'b', 'f', 'g']
letters[:] = []  # -> []
```

- Length

```python
len(letters)  # -> 0
```
Lists

- Lists in lists

```python
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
print(x)  # -> [['a', 'b', 'c'], [1, 2, 3]]
print(x[0][1])  # -> 'b'
```
Lists

- `del` statement

```python
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
print(a)  # -> [1, 66.25, 333, 333, 1234.5]
del a[2:4]
print(a)  # -> [1, 66.25, 1234.5]
del a[:]
print(a)  # -> []
```

- `del` can do more

```python
del a
print(a)  # -> error
```
Tuples

- Similar to lists
- But immutable
- Literals in round parentheses

```python
alist = ['a', 'b', 'c']
atuple = ('a', 'b', 'c')
alist[0] = 'A'  # -> ['A', 'b', 'c']
atuple[0] = 'A'  # -> error
```
Operations over sequences

- Sequence = list, tuple, string, ... and many more

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in s</td>
<td>True if an item of s is equal to x, else False</td>
</tr>
<tr>
<td>x not in s</td>
<td>False if an item of s is equal to x, else True</td>
</tr>
<tr>
<td>s + t</td>
<td>the concatenation of s and t</td>
</tr>
<tr>
<td>s * n or n * s</td>
<td>equivalent to adding s to itself n times</td>
</tr>
<tr>
<td>s[i]</td>
<td>i-th item of s, origin 0</td>
</tr>
<tr>
<td>s[i:j]</td>
<td>slice of s from i to j</td>
</tr>
<tr>
<td>s[i:j:k]</td>
<td>slice of s from i to j with step k</td>
</tr>
<tr>
<td>len(s)</td>
<td>length of s</td>
</tr>
<tr>
<td>min(s)</td>
<td>smallest item of s</td>
</tr>
<tr>
<td>max(s)</td>
<td>largest item of s</td>
</tr>
</tbody>
</table>

See sequences.py
Comparing sequences

- Lexicographically
  - following comparisons are true

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>Sequence 2</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 2, 3)</td>
<td>(1, 2, 4)</td>
<td>&lt;</td>
</tr>
<tr>
<td>[1, 2, 3]</td>
<td>[1, 2, 4]</td>
<td>&lt;</td>
</tr>
<tr>
<td>'ABC'</td>
<td>'C'</td>
<td>&lt;</td>
</tr>
<tr>
<td></td>
<td>'Pascal'</td>
<td>&lt;</td>
</tr>
<tr>
<td></td>
<td>'Python'</td>
<td></td>
</tr>
<tr>
<td>(1, 2, 3, 4)</td>
<td>(1, 2, 4)</td>
<td>&lt;</td>
</tr>
<tr>
<td>(1, 2)</td>
<td>(1, 2, -1)</td>
<td>&lt;</td>
</tr>
<tr>
<td>(1, 2, 3)</td>
<td>(1.0, 2.0, 3.0)</td>
<td>==</td>
</tr>
<tr>
<td>(1, 2, ('aa', 'ab'))</td>
<td>(1, 2, ('abc', 'a'), 4)</td>
<td>&lt;</td>
</tr>
</tbody>
</table>
Conditions in general

- Non-zero number -> true
- Non-empty sequence -> true

```python
a = [1, 2, 3]
print('yes' if a else 'no')  # -> yes

a = []
print('yes' if a else 'no')  # -> no
```

- **and** and **or** – short-circuit evaluation
- no assignment inside expressions (like in C, Java,...)

```python
if (a = get_value()) == 0:  # -> syntax error
    print('zero')
```
set, dict

- **set** – unordered collection of distinct objects
  - literals – {'one', 'two'}

- **frozenset** – immutable set

- **dict** – associative array (hashtable)
  - literals – {'one': 1, 'two': 2, 'three': 3}

See sets_and_dicts.py
Indexing by anything

```
adict = {'one': 1, 'two': 2, 'three': 3}
print(adict['one'])  # -> 1
adict['four'] = 4
print(adict)  # -> {'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Iterating

```
for k, v in adict.items():
    print(k, v)

for k in adict.keys():
    print(k, adict[k])
```
Comprehensions

- a concise way to create lists, sets, dicts
  - this works
    ```python
    squares = []
    for x in range(10):
        squares.append(x**2)
    ```
  - but comprehension is better
    - and shorter, more readable, ..., more Pythonic
    ```python
    squares = [x**2 for x in range(10)]
    ```
- list comprehension
  - brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses
Comprehensions

```
[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
# -> [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4),
#      (3, 1), (3, 4)]
```

- Can be nested

```
# a matrix
m = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
# and a transposed matrix
tm = [[[row[i] for row in m] for i in range(4)]]
```
Comprehensions

• set comprehensions
  • like for lists but in curly braces

```python
word = 'Hello'
letters = {c for c in word}
# another example
a = {x for x in 'abracadabra' if x not in 'abc'}
```

• dict comprehensions
  • also in curly braces but we need to specify both the key and value
    • separated by :

```python
word = 'Hello'
letters = {c: c.swapcase() for c in word}
# -> {'H': 'h', 'e': 'E', 'l': 'L', 'o': 'O'}
```

See comprehensions.py
More collection types

- **bytes**
  - immutable sequences of single bytes

  b'bytes literals are like strings but only with ASCII chars'
  b'escape sequences can be used too\x00'

- **bytearray**
  - mutable counterpart to bytes

See strings_vs_bytes.py
## More collection types

<table>
<thead>
<tr>
<th>Collection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namedtuple</td>
<td>a factory function for creating tuple subclasses with named fields</td>
</tr>
<tr>
<td>deque</td>
<td>a list-like container with fast appends and pops on either end</td>
</tr>
<tr>
<td>ChainMap</td>
<td>a dict-like class for creating a single view of multiple mappings</td>
</tr>
<tr>
<td>Counter</td>
<td>a dict subclass for counting hashable objects</td>
</tr>
<tr>
<td>OrderedDict</td>
<td>a dict subclass that remembers the order entries were added</td>
</tr>
<tr>
<td>defaultdict</td>
<td>a dict subclass that calls a factory function to supply missing values</td>
</tr>
<tr>
<td>heapq</td>
<td>an implementation of the heap queue algorithm</td>
</tr>
</tbody>
</table>

See `other_collections.py`
Naming conventions

- PEP 8, PEP 423

- Classes – camel case
  - `MyBeautifulClass`

- Functions, methods, variables – snake case
  - `my_beautiful_function`, `local_variable`

- “Constants” – capitalized snake case
  - `MAX_VALUE`

- Packages, modules
  - lower case, underscore can be used (discouraged for packages)
  - no conventions as in Java (i.e., like reversed internet name)
  - “pick memorable, meaningful names that aren’t already used on PyPI”
  - The Zen of Python says "Flat is better than nested".
    - two levels is almost always enough

- The Zen of Python
  - `import this`
Special variables/methods of objects

- Many special variables/methods
  - not all objects have all of them
- Naming schema
  - surrounded by double underscores
  - `__name_of_the_special_variable_or_method__`
- `__name__`
  - name of the object
- Others later

```python
import sys
sys.__name__  # -> 'sys'
```