Data structures in Python

Lecture 04

Dr. Colin Rundel

Dictionaries

Dictionaries

Python `dict`s are a *heterogenous*, *ordered* *, *mutable* containers of key value pairs.

Each entry consists of a key (immutable) and a value (anything) - they are designed for the efficient lookup of values using a key.

A `dict` is constructed using `{}` with `:` or via `dict()` using tuples,

```
{'abc': 123, 'def': 456}
```

```
{'abc': 123, 'def': 456}
```

```
dict([('abc', 123), ('def', 456)])
```

```
{'abc': 123, 'def': 456}
```

If all keys are strings then you can assign key value pairs using keyword arguments to `dict()`,

```
dict(hello=123, world=456) # cant use def here as it is reserved
```

```
{'hello': 123, 'world': 456}
```

Allowed key values

keys for a `dict` must be an immutable object (e.g. number, string, or tuple) and keys may be of any type (mutable or immutable).

```
{1: "abc", 1.1: (1,1), "one": ["a","n"], (1,1): lambda x: x**
```

```
{1: 'abc', 1.1: (1, 1), 'one': ['a', 'n'], (1, 1): <function
<lambda> at 0x10948e0c0>}
```

Using a mutable object (e.g. a list) as a key will result in an error,

```
{[1]: "bad"}
```

```
TypeError: unhashable type: 'list'
```

when using a tuple, you need to be careful that all elements are also immutable,

```
{(1, [2]): "bad"}
```

```
TypeError: unhashable type: 'list'
```

dict "subsetting"

The `[]` operator exists for `dict`s but is used for key-based value look ups,

```python
x = {1: 'abc', 'y': 'hello', (1,1): 3.14159}
```

```python
x[1]
```

```
'abc'
```

```python
x['y']
```

```
'hello'
```

```python
x[(1,1)]
```

```
3.14159
```

```python
x[0]
```

```
KeyError: 0
```

```python
x['def']
```

```
KeyError: 'def'
```

Value inserts & replacement

Since dictionaries are mutable, it is possible to insert new key value pairs as well as replace the value associated with an existing key.

```
x = {1: 'abc', 'y': 'hello', (1,1): 3.14159}
```

```
# Insert
x['def'] = -1
x
```

```
{1: 'abc', 'y': 'hello', (1, 1): 3.14159, 'def': -1}
```

```
# Replace
x['y'] = 'goodbye'
x
```

```
{1: 'abc', 'y': 'goodbye', (1, 1): 3.14159, 'def': -1}
```

Removing keys

```
x
```

```
{1: 'abc', 'y': 'goodbye', (1, 1): 3.14159, 'def': -1}
```

```
# Delete
del x[(1,1)]
x
```

```
{1: 'abc', 'y': 'goodbye', 'def': -1}
```

```
x.clear()
x
```

```
{}
```

Other common methods

```python
x = {1: 'abc', 'y': 'hello'}
```

```python
len(x)
```

```
2
```

```python
list(x)
```

```
[1, 'y']
```

```python
tuple(x)
```

```
(1, 'y')
```

```python
1 in x
```

```
True
```

```python
'hello' in x
```

```
False
```

```python
x.keys()
```

```
dict_keys([1, 'y'])
```

```python
x.values()
```

```
dict_values(['abc', 'hello'])
```

```python
x.items()
```

```
dict_items([(1, 'abc'), ('y', 'hello')])
```

```python
x | {(1,1): 3.14159}
```

```
{1: 'abc', 'y': 'hello', (1, 1): 3.14159}
```

```python
x | {'y': 'goodbye'}
```

```
{1: 'abc', 'y': 'goodbye'}
```

Iterating dictionaries

Dictionaries can be used with for loops (and list comprehensions). These loops iterates over the *keys* only, to iterate over the *keys* and *values* use `items()`.

```
for z in {1: 'abc', 'y': 'hello'}:
  print(z)
```

```
1
y
```

```
for k,v in {1: 'abc', 'y': 'hello'}.items()
  print (k,v)
```

```
1 abc
y hello
```

```
[z for z in {1: 'abc', 'y': 'hello'}]
```

```
[1, 'y']
```

```
[(k,v) for k,v in {1: 'abc', 'y': 'hello'}.
```

```
[(1, 'abc'), ('y', 'hello')]
```

Exercise 1

Write a function that takes a two dictionaries as an arguments and merges them into a single dictionary. If there are any duplicate keys, the value from the second dictionary should be used.

```
x = {"a": 1, "b": 2, "c": 3}
y = {"c": 5, "d": 6, "e": 7}

def merge(d1, d2):
  return NULL
```

Sets

Sets

In Python a `set` is a *heterogenous*, *unordered*, *mutable* container of **unique** immutable elements.

A `set` is constructed using `{}` (without using `:`) or via `set()`,

```
{1,2,3,4,1,2}
```

```
{1, 2, 3, 4}
```

```
set((1,2,3,4,1,2))
```

```
{1, 2, 3, 4}
```

```
set("mississippi")
```

```
{'i', 'm', 'p', 's'}
```

All of the elements must be immutable (and therefore hashable),

```
{1,2,[1,2]}
```

```
TypeError: unhashable type: 'list'
```

Subsetting sets

Sets do not use the `[]` operator for element checking or removal,

```
x = set(range(5))
x
```

```
{0, 1, 2, 3, 4}
```

```
x[4]
```

```
TypeError: 'set' object is not subscriptable
```

```
del x[4]
```

```
TypeError: 'set' object doesn't support item deletion
```

Modifying sets

Sets have their own special methods for adding and removing elements,

```python
x = set(range(5))
x
```

```
{0, 1, 2, 3, 4}
```

```python
x.add(9)
x
```

```
{0, 1, 2, 3, 4, 9}
```

```python
x.remove(9)
x.remove(8)
```

```
KeyError: 8
```

```python
x
```

```
{0, 1, 2, 3, 4}
```

```python
x.discard(0)
x.discard(8)
x
```

```
{1, 2, 3, 4}
```

Set operations

```python
x = set(range(5))
x
```

```
{0, 1, 2, 3, 4}
```

```python
3 in x
```

```
True
```

```python
5 in x
```

```
False
```

```python
x.isdisjoint({1,2})
```

```
False
```

```python
x.isdisjoint({5})
```

```
True
```

```python
x <= set(range(6))
```

```
True
```

```python
x.issubset(range(6))
```

```
True
```

```python
x >= set(range(3))
```

```
True
```

```python
x.issuperset(range(3))
```

```
True
```

Set operations (cont)

```
x = set(range(5))
x
```

```
{0, 1, 2, 3, 4}
```

```
x | set(range(10))
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
x.union(range(10))
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
x & set(range(-3,3))
```

```
{0, 1, 2}
```

```
x.intersection(range(-3,3))
```

```
{0, 1, 2}
```

```
x - set(range(2,4))
```

```
{0, 1, 4}
```

```
x.difference(range(2,4))
```

```
{0, 1, 4}
```

```
x ^ set(range(3,9))
```

```
{0, 1, 2, 5, 6, 7, 8}
```

```
x.symmetric_difference(range
```

```
{0, 1, 2, 5, 6, 7, 8}
```

More comprehensions

It is possible to use comprehensions with both `set`s and `dict`s,

```
# Set
{x.lower() for x in "The quick brown fox jumped a lazy dog"}
```

```
{'e', 'd', 'q', 'l', 't', 'u', 'p', 'k', 'f', 'g', 'x', 'm',
'a', 'o', 'y', 'w', 'i', 'c', ' ', 'n', 'b', 'z', 'r', 'h',
'j'}
```

```
# Dict
names = ["Alice", "Bob", "Carol", "Dave"]
grades = ["A", "A-", "A-", "B"]

{name: grade for name, grade in zip(names, grades)}
```

```
{'Alice': 'A', 'Bob': 'A-', 'Carol': 'A-', 'Dave': 'B'}
```

tuple comprehensions

Note that `tuple` comprehensions do not exist,

```
# Not a tuple
(x**2 for x in range(5))
```

```
<generator object <genexpr> at 0x1094dce10>
```

instead you can use a list comprehension which is then cast to a tuple

```
# Is a tuple - via casting a list to tuple
tuple([x**2 for x in range(5)])
```

```
(0, 1, 4, 9, 16)
```

```
tuple(x**2 for x in range(5))
```

```
(0, 1, 4, 9, 16)
```

deques (double ended queue)

are *heterogenous*, *ordered*, *mutable* collections of elements and behave in much the same way as `list`s. They are designed to be efficient for adding and removing elements from the beginning and end of the collection.

These are not part of the base language and are available as part of the built-in `collections` library. We will discuss libraries next time, for now to get access we will import the `deque` function from `collections`.

```
from collections import deque
deque(("A",2,True))
```

```
deque(['A', 2, True])
```

growing and shrinking

```
x = deque(range(3))
x
```

```
deque([0, 1, 2])
```

Values may be added via `.appendleft()` and `.append()` to the beginning and end respectively,

```
x.appendleft(-1)
x.append(3)
x
```

```
deque([-1, 0, 1, 2, 3])
```

values can be removed via `.popleft()` and `.pop()`,

```
x.popleft()
```

```
-1
```

```
x.pop()
```

```
3
```

```
x
```

```
deque([0, 1, 2])
```

`maxlen`

`deque`s can be constructed with an optional `maxlen` argument which determines their maximum size - if this is exceeded values from the opposite side will be dropped.

```
x = deque(range(3), maxlen=4)
x
```

```
deque([0, 1, 2], maxlen=4)
```

```
x.append(0)
x
```

```
deque([0, 1, 2, 0], maxlen=4)
```

```
x.append(0)
x
```

```
deque([1, 2, 0, 0], maxlen=4)
```

```
x.append(0)
x
```

```
deque([2, 0, 0, 0], maxlen=4)
```

```
x.appendleft(-1)
x
```

```
deque([-1, 2, 0, 0], maxlen=4)
```

```
x.appendleft(-1)
x
```

```
deque([-1, -1, 2, 0], maxlen=4)
```

```
x.appendleft(-1)
x
```

```
deque([-1, -1, -1, 2], maxlen=4)
```

Basics of algorithms and data structures

Big-O notation

This is a tool that is used to describe the complexity, usually in time but also in memory, of an algorithm. The goal is to broadly group algorithms based on how their complexity grows as the size of an input grows.

Consider a mathematical function that exactly captures this relationship (e.g. the number of steps in a given algorithm given an input of size n). The Big-O value for that algorithm will then be the largest term involving n in that function.

Complexity	Big-O
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Quasilinear	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

Generally algorithms will vary depending on the exact nature of the data and so often we talk about Big-O in terms of expected complexity and worse case complexity, we also often consider amortization for these worst cases.

Vector / Array

Linked List

Hash table

Time complexity in Python

Operation	list (array)	dict (& set)	deque
Copy	O(n)	O(n)	O(n)
Append	O(1)	—	O(1)
Insert	O(n)	O(1)	O(n)
Get item	O(1)	O(1)	O(n)
Set item	O(1)	O(1)	O(n)
Delete item	O(n)	O(1)	O(n)
`x in s`	O(n)	O(1)	O(n)
`pop()`	O(1)	—	O(1)
`pop(0)`	O(n)	—	O(1)

Exercise 1

For each of the following scenarios, which is the most appropriate data structure and why?

- A fixed collection of 100 integers.
- A queue (first in first out) of customer records.
- A stack (first in last out) of customer records.
- A count of word occurrences within a document.
- The heights of the bars in a histogram with even binwidths

Data structures in R

To tie things back to Sta 523 - the following R objects are implemented using the following data structures.

- Atomic vectors - Array of the given type (int, double, etc.)
- Generic vectors (lists) - Array of SEXPs (R object pointers)
- Environments - Hash map with string-based keys
- Pairlists - Linked list

Environments are not a great option for a general hash map, see `utils::hashtab()` instead.