Lec 04 - Data structures

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

Other containers

Dictionaries

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

Each entry consists of a key (an immutable object) and a value (any object) - they are designed around the efficient lookup of values using a key. More on how this works in a bit.

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

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

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

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

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

if all keys are strings then it is also possible use the key value pairs as keyword arguments to `dict()`,

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

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

Allowed key values

As just mentioned, key values for a `dict` must be an immutable object (number, string, or tuple) and keys do not need to be of a consistent type.

```
{1: "abc", 1.1: (1,1), "one": ["a","n"], (1,1): lambda x: x**2}
```

```
## {1: 'abc', 1.1: (1, 1), 'one': ['a', 'n'], (1, 1): <function <lambda> at 0x108c3db80>}
```

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

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unhashable type: 'list'
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

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

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unhashable type: 'list'
##
```

dict "subsetting"

The `[]` operator exists for `dict`s but is used for value look up using a key,

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

```
x[1]
```

```
## 'abc'
```

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

```
## 'hello'
```

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

```
## 3.14159
```

```
x[0]
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 0
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Value inserts & replacement

As `dict`s are mutable it is possible to insert new key value pairs as well as replace values associated with a 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}
```

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

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

Common methods

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

```
len(x)
```

```
## 2
```

```
list(x)
```

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

```
tuple(x)
```

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

```
1 in x
```

```
## True
```

```
'hello' in x
```

```
## False
```

```
x.keys()
```

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

```
x.values()
```

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

```
x.items()
```

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

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

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

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

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

See more about view objects here

Sets

In Python `set`s are a heterogenous, unordered, mutable containers of unique immutable elements.

`dict`s are constructed using `{}` (without a `:`) 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")
```

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

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

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unhashable type: 'list'
```

Subsetting sets

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

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

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

```
x[4]
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: 'set' object is not subscriptable
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
del x[4]
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: 'set' object doesn't support item dele
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Modifying sets

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

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

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

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

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

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 8
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
x
```

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

Set operations

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

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

```
3 in x
```

```
## True
```

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

```
## False
```

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

```
## True
```

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

```
## True
```

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

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

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

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

```
5 in x
```

```
## False
```

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

```
## True
```

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

```
## True
```

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

```
## True
```

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

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

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

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

more comprehensions

It is possible to use comprehensions with either a `set` or a `dict`,

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

# Dict
```

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

```
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'}
```

Note that `tuple` comprehensions do not exist,

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

# Is a tuple - cast a list to tuple
```

deques (double ended queue)

These 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. More on libraries next time, but to get access we will need to import the library or just the `deque` function from the library.

```
import collections
collections.deque([1,2,3])
```

```
## deque([1, 2, 3])
```

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

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

growing and shrinking

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

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
```

maxlen

deques can be constructed with an options `maxlen` argument which determines their maximum size - if this is exceeded values from the opposite side will be removed.

```
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 space / 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)

Vector / Array

Linked List

Hash table

Time complexity in Python

Operation	list	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)

All of the values presented represented reflect the average Big O time complexity.

Exercise 2

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

- A fixed collection of 100 integers.
- A stack (first in last out) of customer records.
- A queue (first in first out) of customer records.
- A count of word occurances within a document.