Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 11
More Collection Types

Sets, Tuples, Dictionaries, What to use when, Iteration Revisited

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam
ICTP International Centre for Theoretical Physics

Abstract

"All complexities should, if possible, be buried out of sight."
– David J. Wheeler

So far, the only type of collection we learned about are lists.
Python provides more collections, in particular sets, tuples and dictionaries.
Each of these has special properties, making it suitable for particular algorithms and data structures.

Overview

- The `set` type
- The `tuple` type
- The `dict` type
- When to use what
- Iteration Revisited

It's a Python programmer's virtue to know when to use what.

Storing Data Using Sets

- A `set` is an *unordered* collection of *distinct* items
- *Unordered* means the items have no particular order
- An item is in the set or not. That's all.
- *Distincst* means any item appears at most once
- This is how you create a set object:

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> vowels
{'i', 'a', 'o', 'e', 'u'}
```

Notice the set displayed by the shell is unordered.

Storing Data Using Sets

- Duplicates are removed when we create a set:

```
>>> vowels = {'a', 'e', 'a', 'a', 'i', 'o', 'u', 'u'}
>>> vowels
{'i', 'a', 'o', 'e', 'u'}
```

- It might surprise you that the following two sets are equal:

```
>>> {'a', 'e', 'i', 'o', 'u'} ==  {'a', 'e', 'a', 'a', 'i', 'o', 'u', 'u'}
True
```

Two sets are equal if they contain the same items.

Storing Data Using Sets

- By now, it will not surprise you that a `set` is yet another type:

```
>>> type(vowels)
<class 'set'>
>>> type({1, 2, 3})
<class 'set'>
```

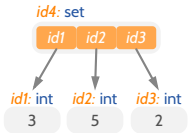
- To create an empty `set` we have to do this:

```
>>> set()
set()
>>> type(set())
<class 'set'>
```

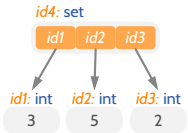
It will soon become clear why we can't use `{}` for an empty set.

The Set Memory Model

```
set([2, 3, 5])
```




```
set([2, 3, 5, 5, 2, 3])
```



Constructing a `set` from a `list` removes duplicates.

More on Creating Sets

- The function `set` takes at most one argument:

```
>>> set(2, 3, 4)
Traceback (most recent call last):
  File ``<stdin>'', line 1, in <module>
TypeError: set expected at most 1 arguments, got 3
```

- A set can be created from another set:

```
>>> vowels = {'a', 'e', 'a', 'a', 'i', 'o', 'u', 'u'}
>>> vowels
{'i', 'a', 'o', 'e', 'u'}
>>> set({5, 3, 1})
{1, 3, 5}
```

- Or from a `range` (or any other generator):

```
>>> set(range(5))
{0, 1, 2, 3, 4}
```

Any sequence works. There are some we haven't seen yet.

Set Operations

- Python provides set operations known from maths like intersection, add and remove
- The set operations are implemented as methods
- As usual, you can read all about them in the documentation

https://docs.python.org/3.6/library/stdtypes.html#set

- Or by using the `help` function:

```
>>> help(set)
```

Many set operations also have corresponding operators.

Set Operations

- Sets are mutable, that means we can change the value of set objects
- In particular, we can add or remove items:

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> vowels
{'i', 'a', 'o', 'e', 'u'}
```

```
>>> vowels.add('y')
>>> vowels
{'o', 'i', 'a', 'e', 'u', 'y'}
```

```
>>> vowels.remove('e')
>>> vowels
{'o', 'i', 'a', 'u', 'y'}
```

```
>>> vowels.clear()
>>> vowels
set()
```

Note again that sets are *unordered*.

Set Methods & Operators

Method Call	Operator
`set1.difference(set2)`	`set1 - set2`
`set1.intersection(set2)`	`set1 & set2`
`set1.issubset(set2)`	`set1 <= set2`
`set1.issuperset(set2)`	`set1 >= set2`
`set1.union(set2)`	`set1 | set2`
`set1.symmetric_difference(set2)`	`set1 ^ set2`

All of these create new objects.

What Do Set Operations do for Us?

- Let's start from this:

```
>>> lows = set([0, 1, 2, 3, 4])
>>> odds = set([1, 3, 5, 7, 9])
```

- And now let's see what the method calls an operators do:

```
>>> lows.difference(odds)
{0, 2, 4}
>>> lows.intersection(odds)
{1, 3}
>>> lows.issubset(odds)
False
>>> lows.issuperset(odds)
False
>>> lows.union(odds)
{0, 1, 2, 3, 4, 5, 7, 9}
>>> lows.symmetric_difference(odds)
{0, 2, 4, 5, 7, 9}
```

```
>>> lows - odds
{0, 2, 4}
>>> lows & odds
{1, 3}
>>> lows <= odds
False
>>> lows >= odds
False
>>> lows | odds
{0, 1, 2, 3, 4, 5, 7, 9}
>>> lows ^ odds
{0, 2, 4, 5, 7, 9}
```

Remember the operators just call the methods.

Iterating Over Sets

- Using this set:

```
>>> lows = set([0, 1, 2, 3, 4])
```

- We can do the following:

```
>>> for low in lows:
...     print(low)
...
0
1
2
3
4
```

Iteration with `for` works for all sequences.

Storing Data Using Tuples

- Tuples are *immutable ordered* sequences
- They are created as follows:

```
>>> bases = ('A', 'C', 'G', 'T')
>>> bases
('A', 'C', 'G', 'T')
>>> numbers = tuple([1, 2, 3])
(1, 2, 3)
```

- Tuples can be empty:

```
>>> type(())
<class 'tuple'>
```

- One-item tuples require special syntax:

```
>>> type((8))
<class 'int'>
>>> type((8,))
<class 'tuple'>
```

You can think of tuples as frozen lists.

Storing Data Using Tuples

- We can use indices on tuples:

```
>>> bases = ('A', 'C', 'G', 'T')
>>> bases[1]
'C'
```

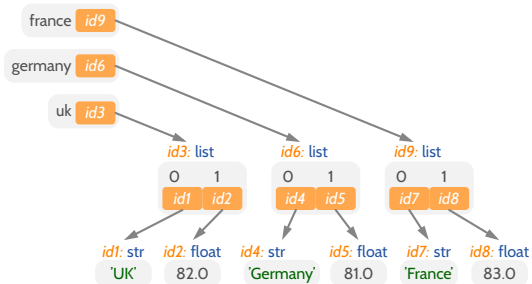
- We can iterate over tuples:

```
>>> bases = ('A', 'C', 'G', 'T')
>>> for base in bases:
...     print(base)
...
A
C
G
T
```

Indexing does only work for ordered sequences.

Tuples & Mutability: The Memory Model

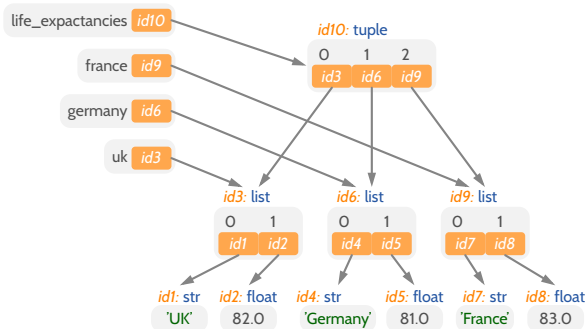
```
>>> uk = ['UK', 82.0]
>>> germany = ['Germany', 81.0]
>>> france = ['France', 83.0]
```



There is no `tuple` yet – we'll construct it from the variables.

Tuples & Mutability: The Memory Model

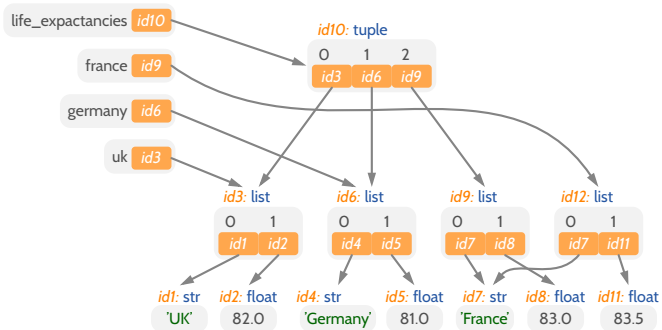
```
>>> life_expectancies = (uk, germany, france)
>>> life_expectancies
(['UK', 82.0], ['Germany', 81.0], ['France', 83.0])
```



The `tuple` contains references to objects, not variables.

Tuples & Mutability: The Memory Model

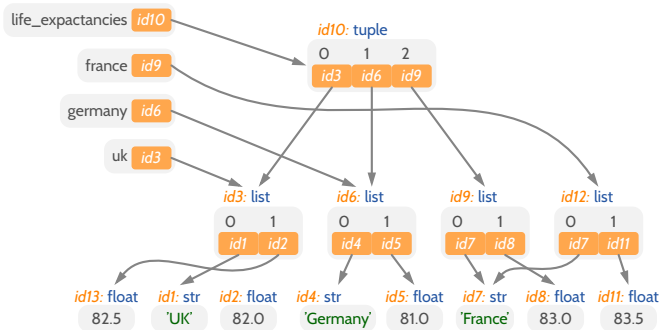
```
>>> france = ['France', 83.5]
>>> life_expectancies
(['UK', 82.0], ['Germany', 81.0], ['France', 83.0])
```



Note that the tuple has not changed.

Tuples & Mutability: The Memory Model

```
>>> life_expectancies[0][1] = 82.5
>>> uk
['UK', 82.5]
```



The `tuple` is immutable. But the first item, a `list`, is mutable.

Unpacking

- Python provides an elegant way for *unpacking* sequences

```
>>> uk, germany, france = life_expectancies
>>> germany
['Germany', 81.0]
```

```
>>> coordinates = [1.2, 0.0, 7.32]
>>> x, y, z = coordinates
>>> z
7.32
```

```
>>> print(*coordinates)
```

This works for all sequences.

More Elegant Loops

- Tuples are useful for processing sequences in parallel
- Python provides the built-in function `zip`:

```
metals = ['Li', 'Na', 'K']
weights = [6.941, 22.98976928, 39.0983]
```

```
for items in zip(metals, weights):
    print(items[0], items[1])
```

```
for metal, weight in zip(metals, weights):
    print(metal, weight)
```

- The function `enumerate` provides indices *and* items:

```
for idx, metal in enumerate(metals):
    print(idx, metal)
```

Use these features. They are more readable and robust than raw indices.

Storing Data Using Dictionaries

- Suppose we want to count the number of observations for each bird:

```
bird_counts = []

for line in observations_file:
    bird = line.strip()
    found = False

    # Find bird in the list of bird counts.
    for entry in bird_counts:
        if entry[0] ==  bird:
            entry[1] = entry[1] + 1
            found = True
    if not found:
        bird_counts.append([bird, 1])

for entry in bird_counts:
    print(entry[0], entry[1])
```

observations.txt

```
canada goose
canada goose
long-tailed jaeger
canada goose
snow goose
canada goose
long-tailed jaeger
canada goose
northern fulmar
```

This works. What do you think is not nice about it?

Storing Data Using Dictionaries

- It would be nice if we could label items with something else than an index
- Python provides *dictionaries* for exactly that purpose:

```
>>> months = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4}
>>> months
{'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4}
>>> type(months)
<class 'dict'>
```

- Dictionary *values* are accessed via *keys*

```
>>> months['Feb']
2
```

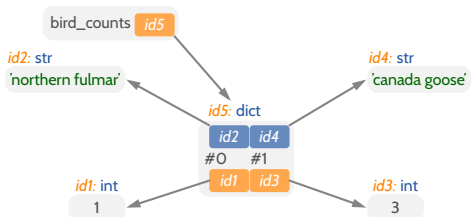
- As usual, we can create empty dictionaries:

```
>>> type({})
<class 'dict'>
```

Dictionaries are mutable.

Dictionaries: The Memory Model

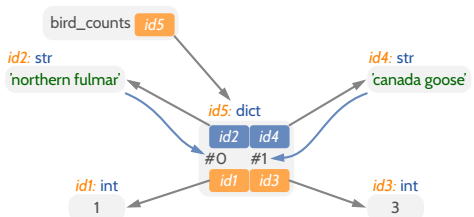
```
>>> bird_counts = {'canada goose': 3, 'northern fulmar': 1}
>>> bird_counts
{'northern fulmar': 1, 'canada goose': 3}
```



A dictionary has parallell lists of *keys* and *values*.

Dictionaries: The Memory Model

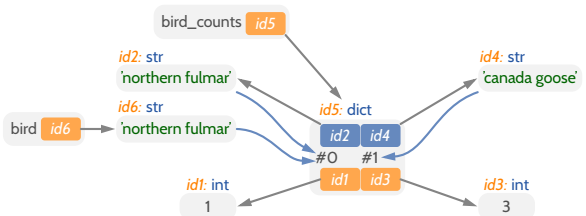
```
>>> bird_counts = {'canada goose': 3, 'northern fulmar': 1}
>>> bird_counts['canada goose']
3
```



A dictionary maps *keys* to *values* using *hashes*.

Dictionaries: The Memory Model

```
>>> bird = 'northern fulmar'
>>> bird_counts[bird]
1
```



The key lookup is based on a *hash* of the key's value, *not* the key object.

Updating and Checking for Membership

- Suppose we made a mistake:

```
>>> months = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':5}
>>> months
{'Jan':1, 'Feb':2, 'Mar':3, 'Apr':5}
>>> months['Apr'] = 4
>>> months
{'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4}
```

- The **`in`** operator checks for the presence of a key:

```
>>> 'Jan' in months
True
>>> 'May' in months
False
```

- We can update and retrieve items at the same time:

```
>>> months.get('May', 5)
5
>>> 'May' in months
True
```

Dictionary Operation Examples

- We can get a list of the keys:

```
>>> months.keys()
['Jan', 'Feb', 'Mar', 'Apr', 'May']
```

- And of the items:

```
>>> months.items()
[1, 2, 3, 4, 5]
```

- Removing all keys and items:

```
>>> months.clear()
>>> months
{}
```

Check the documentations for more.

Looping Over Dictionaries

- Looping over a dictionary works like with any other collection:

```
>>> for m in months:
...     print(m, months[m])
...
Jan 1
Feb 2
Mar 3
Apr 4
May 5
```

Remember Python is all about how things behave.

Storing Data Using Dictionaries

- Suppose we want to count the number of observations for each bird:

```
bird_counts = []

for line in observations_file:
    bird = line.strip()
    found = False

    # Find bird in the list of bird counts.
    for entry in bird_counts:
        if entry[0] ==  bird:
            entry[1] = entry[1] + 1
            found = True
    if not found:
        bird_counts.append([bird, 1])

for entry in bird_counts:
    print(entry[0], entry[1])
```

observations.txt

```
canada goose
canada goose
long-tailed jaeger
canada goose
snow goose
canada goose
long-tailed jaeger
canada goose
northern fulmar
```

This works. What do you think is not nice about it?

Storing Data Using Dictionaries

- Suppose we want to count the number of observations for each bird:

```
bird_counts = {}

for line in observations_file:
    bird = line.strip()
    bird_counts[bird] = bird_counts.get(bird, 0) + 1

for bird in bird_counts:
    print(bird, bird_counts[bird])
```

observations.txt

```
canada goose
canada goose
long-tailed jaeger
canada goose
snow goose
canada goose
long-tailed jaeger
canada goose
northern fulmar
```

This is *much* better!

Exercises Lecture 11