Week 8 - Lecture 15

István Albert
BMB, Bioinformatics
Dictionary – storage that permits very fast retrieval by a key

• Lists store objects at integer indices:
  \[
  \text{data[1]} = 10
  \]

• Sets store unique elements

• Dictionaries store values for keys:
  \[
  \text{data[“John”]} = 10
  \]
A listing of the data structures

```python
# data structures

a = list()
b = tuple()
c = set()
d = dict()

for x in (a, b, c, d):
    print(type(x), x)
```

```
<type 'list'> []
<type 'tuple'> ()
<type 'set'> set([])
<type 'dict'> {}
```
Shorthand notations for datatypes

```python
# shorthand notations for data structures
# no shorthand for set? awwww, let's give it a hug

a = []
b = ()
c = set()
d = {}

for x in (a, b, c, d):
    print(type(x), x)
```

```
<type 'list'> []
<type 'tuple'> ()
<type 'set'> set([])
<type 'dict'> {}
```
Creating dictionaries \rightarrow \{ \}

```python
data = {}
vals = ["A", "B", "C"]

for key in vals:
data[key] = key + key + key

print(data)
print("The value for \(C\) is", data["C"])
```

```
{'A': 'AAA', 'C': 'CCC', 'B': 'BBB'}
The value for \(C\) is CCC
```
Pre-declare populated dictionaries

data = { "A":100, "B":200, "C":300 }

print type(data), data
print data["A"]
print data["B"]
Dictionary methods

```
data = { "A":100, "B":200, "C":300 }
print data.keys()
print data.values()
print data.items()
```

Note the order! Keys and values have the same “unexpected” order.
Dictionary order

• Keys and values will be returned in an “undetermined” order

• The order keys and values is the same.

• Order is not random ➔ but it depends on the method of construction (same as set).

   If you need order ➔ use lists
You may build a dictionary step by step

```python
dict = {'Bob': 25, 'Alice': 20, 'Zoe': 30}
Bob's age is 25
```
But you should try building it with collated data

```python
def dict_create1.py
names = ['Alice', 'Bob', 'Zoe']
ages = [20, 25, 30]
collate = zip(names, ages)
data = dict(collate)
print "Collate", collate
print "Dict = ", data
print "Bob's age is ", data["Bob"]
```
Inverse collation
compare to previous slide
What can be stored as keys, values?

- Only **immutable objects** may be keys → strings, numbers, tuples, immutable sets

- **Anything** can be a value → string, list, set or any other object

- The **keys** in a dictionary are unique (like a set),
Creating a dictionary from the data

```python
import bmbb

fname = 'GPL9270/GSM455823.txt'
gen = bmbb.getcolumn(fname, 'ID_REF')
vals = bmbb.getfloats(fname, 'CH1_SIG_MEAN')
coll = zip(genes, vals)
store = dict(coll)

# print coll[:10]
print store['tetB']
```

```
7190.0
```
Common dictionary methods

```python
import bmbb, glob

fname = 'GPL9270/GSM455823.txt'
genes = bmbb.getcolumn(fname, 'ID_REF')
vals = bmbb.getfloats(fname, 'CH1_SIG_MEAN')
# cut it down for this example
coll = zip(genes, vals)[:3]
store = dict(coll)

print store.keys()
print store.values()
print store.items()
```

```python
['ssuA', 'yccK', 'opuCD']
[5298.0, 7754.0, 12582.0]
[('ssuA', 5298.0), ('yccK', 7754.0), ('opuCD',
```
Iterating over a dictionary

```python
# explicit iteration over keys or values
for key in store.keys():
    print key

for value in store.values():
    print value

for key, value in store.items():
    print key, value
```
Homework 15: create the data named x (at the beginning of the program) that will print as follows

```python
print x['red']
print len(x['blue'])
print x['pink']['size']
print x['green'] & x['brown']
print len(x)
```

Define x here
Week 8 - Lecture 16

István Albert
BMB, Bioinformatics
Stop the unwanted services – very few processes do actually need to run all the time.
Resources and scaling

• It is very important to develop a sense of how long processes “should” take

• Understand how computational resources need to scale with data size

• There are always bottlenecks where the program spends 90% of time whereas the rest is plenty fast
Resource types

- **CPU performance**: both frequency of operations (~ giga-hertz), instruction size (how much work per operation), and number of CPUs
- **Memory size**: amount of data that can be loaded up at any given time (~ Giga-bytes)
- **Storage size**: the size of data that can be stored on the computer (~ Terra-bytes)

More esoteric resources:

- **Network speed**: how much data can be transferred across computers (typically there is difference between upload and download speeds)
- **Hard drive speed**: how much data can be transferred from a hard disk to the memory
- **Data bus speed**: how much data can be transferred from the memory to the CPU
```python
import time

def square(x):
    return x**x

start = time.time()
data = range(10**5)
data = map(square, data)
end = time.time()

print "Done in %.4s seconds" % (end-start)
```

```
Done in 0.02 seconds
```
Storage

• The basic unit of storage is a byte → 8 bits

• Most data types may take power of 2 bytes

  4 bytes → 32 bit integers
  8 bytes → 64 bit integers

A single string character → one byte

  1 → 4 bytes
  ‘1’ → 1 byte
  10000000 → 4 bytes
  ‘10000000’ → 8 bytes
  “Hello World!” → 12 bytes

The minimal storage is at least the **data size** * **data count** but when the data is stored in a container

**apparent data size = (data size + container overhead)**
Memory benchmarking

Built-in tools: Windows Task Manager or Activity Monitor on a Mac. You can download other tools.
Memory usage measurements for 20, 30, 40, 50 million values

Note the slopes.

50 million values lead to memory error
How many values do we actually store?

```python
import time

def square(x):
    return x**x

data = range(5*10**7)
data = map(square, data)
end = time.sleep(5)

print "Done"
```

50 million

another 50 million

100 million at the end of the map procedure.

The solution to problem are iterators. We’ll cover these later.
Representation differences

```python
import time

data = range(4*10**7)
print "Integers"
time.sleep(5)

vals = map(float, data)
print "Floats"
time.sleep(5)
print "Done"
```

Start with smaller values as your computer may have less memory
**Algorithm efficiency**

Big O notation characterizes functions by their growth → but only keeps the largest factor.

\[ n \rightarrow \text{the number of elements} \]

A few simple problems in Big O notations. If the solution requires us to look at only one element then we say that the algorithm is of complexity \( O(1) \)

- If you don’t need to go through all elements → \( O(1) \)
- If we have to look at all elements → \( O(n) \)
- If we have to look at all elements twice it still stays → \( O(n) \)
- If we have to look at \( n(n-1) \) elements → \( O(n^2) \)
- Matrix multiplication → \( O(n^3) \)
Testing data structures for efficiency

A in B operator returns the truth value of element A being present in container B
Repeat the test for more reliable results
Built-in timer module helps timings

```python
from timeit import Timer

setup = "data = range(1000)"
stmt = "1 in data"

if __name__ == '__main__':
    t = Timer(setup=setup, stmt=stmt)
    print t.timeit(number=100)
```

`python -t C:\cygwin\home\jalbert\docs\jalbert-web\ppt\fall-2010\597D\week8\timeit-ex.py 2>&1` returned 0

8.70344160861e-06
Compare the behavior for “best” and “worst” case.

```python
def main():
    setup = "data = range(1000)"
    stmt1 = "1 in data"
    stmt2 = "'Joe' in data"

    if __name__ == '__main__':
        t1 = Timer(setup=setup, stmt=stmt1)
        t2 = Timer(setup=setup, stmt=stmt2)
        print t1.timeit(number=100)
        print t2.timeit(number=100)

def main():
    setup = "data = range(1000)"
    stmt1 = "1 in data"
    stmt2 = "'Joe' in data"

    if __name__ == '__main__':
        t1 = Timer(setup=setup, stmt=stmt1)
        t2 = Timer(setup=setup, stmt=stmt2)
        print t1.timeit(number=100)
        print t2.timeit(number=100)
```
• Estimate the memory usage necessary for storing a large number the same data as integers, floats and strings. Try to find the largest number that you can comfortably fit into memory. See if the result is different if you generate numbers not from 0 but from 10000 and up. Find the ratios of the required storage relative to the smallest (i.e. type x requires 10% more space than the smallest)

• Estimate the memory usage necessary for storing the same integers as a list, a set and a dictionary. Try a smaller and a larger number (twice as many) integers. Find the ratios of the required storage relative to the smallest (type x requires 10% more space than the smallest).

• Estimate the algorithmic efficiency of the ‘in’ operator relative to a list(), a set() and a dictionary(). Create three sizes of data for each and test the speed of testing for containment separately for the first and last items. Which one is the fastest and which one is the slowest. Can you estimate the O(?) scaling?