Dictionaries

A “Good morning” dictionary

English: Good morning
Spanish: Buenas días
Swedish: God morgon
German: Guten morgen
Venda: Ndi matscheloni
Afrikaans: Goeie môre

What's a dictionary?

A dictionary is a table of items.
Each item has a "key" and a "value"

Keys	Values
English	Good morning
Spanish	Buenas días
Swedish	God morgon
German	Guten morgen
Venda	Ndi matscheloni
Afrikaans	Goeie môre

Look up a value

I want to know "Good morning" in Swedish.

Step 1: Get the "Good morning" table

Keys	Values
English	Good morning
Spanish	Buenas días
Swedish	God morgon
German	Guten morgen
Venda	Ndi matscheloni
Afrikaans	Goeie môre

Find the item

Step 2: Find the item where the key is “Swedish”

Keys	Values
English	Good morning
Spanish	Buenas días
Swedish	God morgon
German	Guten morgen
Venda	Ndi matscheloni
Afrikaans	Goeie môre

Get the value

Step 3: The value of that item is how to say "Good morning" in Swedish -- "God morgon"

Keys	Values
English	Good morning
Spanish	Buenas días
Swedish	God morgon
German	Guten morgen
Venda	Ndi matscheloni
Afrikaans	Goeie môre

In Python

```
>>> good_morning_dict = {
...     "English": "Good morning",
...     "Swedish": "God morgon",
...     "German": "Guten morgen",
...     "Venda": "Ndi matscheloni",
... }
>>> print good_morning_dict["Swedish"]
God morgon
>>>
```

(I left out Spanish and Afrikaans because they use ‘special’ characters. Those require Unicode, which I’m not going to cover.)

Dictionary examples

An empty dictionary

```
>>> D1 = {}
>>> len(D1)
0
```

A dictionary with 2 items

```
>>> D2 = {"name": "Andrew", "age": 33}
>>> len(D2)
2
>>> D2["name"]
'Andrew'
>>> D2["age"]
33
```

Keys are case-sensitive

```
>>> D2["AGE"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'AGE'
>>>
```

Add new elements

```
>>> my_sister = {}
>>> my_sister["name"] = "Christy"
>>> print "len =", len(my_sister), "and value is", my_sister
len = 1 and value is {'name': 'Christy'}
>>> my_sister["children"] = ["Maggie", "Porter"]
>>> print "len =", len(my_sister), "and value is", my_sister
len = 2 and value is {'name': 'Christy', 'children': ['Maggie', 'Porter']}
>>>
```

Get the keys and values

```
>>> city = {"name": "Cape Town", "country": "South Africa",
...     "population": 2984000, "lat.": -33.93, "long.": 18.46}
>>> print city.keys()
['country', 'long.', 'lat.', 'name', 'population']
>>> print city.values()
['South Africa', 18.460000000000001, -33.93, 'Cape Town', 2984000]
>>> for k in city:
...     print k, "=", city[k]
...
country = South Africa
long. = 18.46
lat. = -33.93
name = Cape Town
population = 2984000
>>>
```

A few more examples

```
>>> D = {"name": "Johann", "city": "Cape Town"}
>>> counts["city"] = "Johannesburg"
>>> print D
{'city': 'Johannesburg', 'name': 'Johann'}
>>> del counts["name"]
>>> print D
{'city': 'Johannesburg'}
>>> counts["name"] = "Dan"
>>> print D
{'city': 'Johannesburg', 'name': 'Dan'}
>>> D.clear()
>>>
>>> print D
{}
>>>
```

Ambiguity codes

Sometimes DNA bases are ambiguous.

Eg, the sequencer might be able to tell that a base is not a G or T but could be either A or C.

The standard (IUPAC) one-letter code for DNA includes letters for ambiguity.

M is A or C	Y is C or T	D is A, G or T
R is A or G	K is G or T	B is C, G or T
W is A or T	V is A, C or G	N is G, A, T or C
S is C or G	H is A, C or T	

Count Bases #1

This time we'll include all 16 possible letters

```
>>> seq = "TKKAMRCRAATARKWC"
>>> A = seq.count("A")
>>> B = seq.count("B")
>>> C = seq.count("C")
>>> D = seq.count("D")
>>> G = seq.count("G")
>>> H = seq.count("H")
>>> K = seq.count("K")
>>> M = seq.count("M")
>>> N = seq.count("N")
>>> R = seq.count("R")
>>> S = seq.count("S")
>>> T = seq.count("T")
>>> V = seq.count("V")
>>> W = seq.count("W")
>>> Y = seq.count("Y")
>>> print "A =", A, "B =", B, "C =", C, "D =", D, "G =", G, "H =", H, "K =", K, "M
=", M, "N =", N, "R =", R, "S =", S, "T =", T, "V =", V, "W =", W, "Y =", Y
A = 4 B = 0 C = 2 D = 0 G = 0 H = 0 K = 3 M = 1 N = 0 R = 3 S = 0
T = 2 V = 0 W = 1 Y = 0
>>>
```

Don't do this!

Let the computer help out

Count Bases #2

Using a dictionary

```
>>> seq = "TKKAMRCRAATARKWC"
>>> counts = {}
>>> counts["A"] = seq.count("A")
>>> counts["B"] = seq.count("B")
>>> counts["C"] = seq.count("C")
>>> counts["D"] = seq.count("D")
>>> counts["G"] = seq.count("G")
>>> counts["H"] = seq.count("H")
>>> counts["K"] = seq.count("K")
>>> counts["M"] = seq.count("M")
>>> counts["N"] = seq.count("N")
>>> counts["R"] = seq.count("R")
>>> counts["S"] = seq.count("S")
>>> counts["T"] = seq.count("T")
>>> counts["V"] = seq.count("V")
>>> counts["W"] = seq.count("W")
>>> counts["Y"] = seq.count("Y")
>>> print counts
{'A': 4, 'C': 2, 'B': 0, 'D': 0, 'G': 0, 'H': 0, 'K': 3, 'M': 1,
'N': 0, 'S': 0, 'R': 3, 'T': 2, 'W': 1, 'V': 0, 'Y': 0}
>>>
```

Don’t do this either!

Count Bases #3

use a `for` loop

```
>>> seq = "TKKAMRCRAATARKWC"
>>> counts = {}
>>> for letter in "ABCDGHKMNRSTVWY":
...     counts[letter] = seq.count(letter)
...
>>> print counts
{'A': 4, 'C': 2, 'B': 0, 'D': 0, 'G': 0, 'H': 0, 'K': 3, 'M': 1, 'N': 0, 'S': 0, 'R':
3, 'T': 2, 'W': 1, 'V': 0, 'Y': 0}
>>> for base in counts.keys():
...     print base, "=", counts[base]
...
A = 4
C = 2
B = 0
D = 0
G = 0
H = 0
K = 3
M = 1
N = 0
S = 0
R = 3
T = 2
W = 1
V = 0
Y = 0
>>>
```

Count Bases #4

Suppose you don't know all the possible bases.

If the base isn't a key in the counts dictionary then use zero. Otherwise use the value from the dict

```
>>> seq = "TKKAMRCRAATARKWC"
>>> counts = {}
>>> for base in seq:
...     if base not in counts:
...             n = 0
...     else:
...             n = counts[base]
...     counts[base] = n + 1
...
>>> print counts
{'A': 4, 'C': 2, 'K': 3, 'M': 1, 'R': 3, 'T': 2, 'W': 1}
>>>
```

Count Bases #5 (Last one!)

The idiom "use a default value if the key doesn't exist" is very common. Python has a special method to make it easy.

```
>>> seq = "TKKAMRCRAATARKWC"
>>> counts = {}
>>> for base in seq:
...   counts[base] = counts.get(base, 0) + 1
...
>>> print counts
{'A': 4, 'C': 2, 'K': 3, 'M': 1, 'R': 3, 'T': 2, 'W': 1}
>>> counts.get("A", 9)
4
>>> counts["B"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'B'
>>> counts.get("B", 9)
9
>>>
```

Reverse Complement

```
>>> complement_table = {"A": "T", "T": "A", "C": "G", "G": "C"}
>>> seq = "CCTGTATT"
>>> new_seq = []
>>> for letter in seq:
...     complement_letter = complement_table[letter]
...     new_seq.append(complement_letter)
...
>>> print new_seq
['G', 'G', 'A', 'C', 'A', 'T', 'A', 'A']
>>> new_seq.reverse()
>>> print new_seq
['A', 'A', 'T', 'A', 'C', 'A', 'G', 'G']
>>> print "".join(new_seq)
AATACAGG
>>>
```

Listing Codons

```
>>> seq = "TCTCCAAGACGCATCCCAGTG"
>>> seq[0:3]
'TCT'
>>> seq[3:6]
'CCA'
>>> seq[6:9]
'AGA'
>>> range(0, len(seq), 3)
[0, 3, 6, 9, 12, 15, 18]
>>> for i in range(0, len(seq), 3):
...     print "Codon", i/3, "is", seq[i:i+3]
...
Codon 0 is TCT
Codon 1 is CCA
Codon 2 is AGA
Codon 3 is CGC
Codon 4 is ATC
Codon 5 is CCA
Codon 6 is GTG
>>>
```

The last "codon"

```
>>> seq = "TCTCCAA"
>>> for i in range(0, len(seq), 3):
...     print "Base", i/3, "is", seq[i:i+3]
...
Base 0 is TCT
Base 1 is CCA
Base 2 is A
>>>
```

Not a codon!

What to do? It depends on what you want. But you'll probably want to know if the sequence length isn't divisible by three.

The ‘%’ (remainder) operator

```
>>> 0 % 3
0
>>> 1 % 3
1
>>> 2 % 3
2
>>> 3 % 3
0
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>>
```

```
>>> seq = "TCTCCAA"
>>> len(seq)
7
>>> len(seq) % 3
1
>>>
```

Two solutions

First one -- refuse to do it

```
if len(seq) % 3 != 0:  # not divisible by 3
     print "Will not process the sequence"
else:
     print "Will process the sequence"
```

Second one -- skip the last few letters
Here I'll adjust the length

```
>>> seq = "TCTCCAA"
>>> for i in range(0, len(seq) - len(seq)%3, 3):
...     print "Base", i/3, "is", seq[i:i+3]
...
Base 0 is TCT
Base 1 is CCA
>>>
```

Counting codons

```
>>> seq = "TCTCCAAGACGCATCCCAGTG"
>>> codon_counts = {}
>>> for i in range(0, len(seq) - len(seq)%3, 3):
...     codon = seq[i:i+3]
...     codon_counts[codon] = codon_counts.get(codon, 0) + 1
...
>>> codon_counts
{'ATC': 1, 'GTG': 1, 'TCT': 1, 'AGA': 1, 'CCA': 2, 'CGC': 1}
>>>
```

Notice that the codon_counts dictionary elements aren't sorted?

Sorting the output

People like sorted output. It's easier to find "GTG" if the codon table is in order.

Use `keys` to get the dictionary keys then use `sort` to sort the keys (put them in order).

```
>>> codon_counts = {'ATC': 1, 'GTG': 1, 'TCT': 1, 'AGA': 1, 'CCA': 2, 'CGC': 1}
>>> codons = codon_counts.keys()
>>> print codons
['ATC', 'GTG', 'TCT', 'AGA', 'CCA', 'CGC']
>>> codons.sort()
>>> print codons
['AGA', 'ATC', 'CCA', 'CGC', 'GTG', 'TCT']
>>> for codon in codons:
...     print codon, "=", codon_counts[codon]
...
AGA = 1
ATC = 1
CCA = 2
CGC = 1
GTG = 1
TCT = 1
>>>
```

Exercise 1 - letter counts

Ask the user for a sequence. The sequence may include ambiguous codes (letters besides A, T, C or G). Use a dictionary to find the number of times each letter is found.

Note: your output may be in a different order than mine.

Test case #1

```
Enter DNA: ACRSAS
A = 2
C = 1
R = 2
S = 2
```

Test case #2

```
Enter DNA: TACATCGATGCWACTN
A = 4
C = 4
G = 2
N = 1
T = 4
W = 1
```

Exercise 2

Modify your program from Exercise 1 to find the length and letter counts for each sequence in /usr/coursehome/dalke/ambiguous_sequences.seq
It is okay to print the base counts in a different order.

The first three sequences

```
Sequence has 1267 bases
A = 287
C = 306
B = 1
G = 389
R = 1
T = 282
Y = 1
Sequence has 553 bases
A = 119
C = 161
T = 131
G = 141
N = 1
Sequence has 1521 bases
A = 402
C = 196
T = 471
G = 215
N = 237
```

The last three sequences

```
Sequence has 1285 bases
A = 327
Y = 1
C = 224
T = 371
G = 362
Sequence has 570 bases
A = 158
C = 120
T = 163
G = 123
N = 6
Sequence has 1801 bases
C = 376
A = 465
S = 1
T = 462
G = 497
```

Exercise 3

Modify your program from Exercise 2 so the base counts are printed in alphabetical order. (Use the `keys` method of the dictionary to get a list, then use the `sort` method of the list.)

The first sequence output should write

```
Sequence has 1267 bases
A = 287
B = 1
C = 306
G = 389
R = 1
T = 282
Y = 1
```

Exercise 4

Write a program to count the total number of bases in all of the sequences in the file /usr/coursehome/dalke/ambiguous_sequences.seq and the total number of each base found, in order

Here's what I got. Am I right?

```
File has 24789 bases
A = 6504
B = 1
C = 5129
D = 1
G = 5868
K = 1
M = 1
N = 392
S = 2
R = 3
T = 6878
W = 1
Y = 8
```

Exercise 5

Do the same as exercise 4 but this time use /coursehome/dalke/sequences.seq

Compare your results with someone else.

Then try /coursehome/dalke/many_sequences.seq

Compare results then compare how long it took the program to run. (See note on next page.)

How long did it run?

You can ask Python for the current time using the `datetime` module we talked about last week.

```
>>> import datetime
>>> start_time = datetime.datetime.now()
>>> # put the code to time in here
>>> end_time = datetime.datetime.now()
>>> print end_time - start_time
0:00:09.335842
>>>
```

This means it took me 9.3 seconds to write the third and fourth lines.

Exercise 6

Write a program which prints the reverse complement of each sequence from the file /coursehome/dalke/10_sequences.seq

This file contains only A, T, C, and G letters.

Exercise 7

Modify the program from Exercise 6 to find the reverse complement of an ambiguous DNA sequence.
(See next page for the data table.)
Test it against /coursehome/dalke/sequences.seq
Compare your results with someone else.

To do that, run the program from the unix shell and have it save your output to a file. Compare using 'diff'.

```
python your_file.py > output.dat
diff output.dat /coursehome/surname/output.dat
```

Ambiguous complements

```
ambiguous_dna_complement = {
    "A": "T",
    "C": "G",
    "G": "C",
    "T": "A",
    "M": "K",
    "R": "Y",
    "W": "W",
    "S": "S",
    "Y": "R",
    "K": "M",
    "V": "B",
    "H": "D",
    "D": "H",
    "B": "V",
    "N": "N",
    }
```

This is also the file

/coursehome/dalke/complements.py

Translate DNA into protein

Write a program to ask for a DNA sequence. Translate the DNA into protein. (See next page for the codon table to use.) When the codon doesn't code for anything (eg, stop codon), use "*". Ignore the extra bases if the sequence length is not a multiple of 3. Decide how you want to handle ambiguous codes.

Come up with your own test cases. Compare your results with someone else or with a web site.

Standard codon table

This is also in the file /usr/coursehome/dalke/codon_table.py

```
table = {
     'TTT': 'F', 'TTC': 'F', 'TTA': 'L', 'TTG': 'L', 'TCT': 'S',
     'TCC': 'S', 'TCA': 'S', 'TCG': 'S', 'TAT': 'Y', 'TAC': 'Y',
     'TGT': 'C', 'TGC': 'C', 'TGG': 'W', 'CTT': 'L', 'CTC': 'L',
     'CTA': 'L', 'CTG': 'L', 'CCT': 'P', 'CCC': 'P', 'CCA': 'P',
     'CCG': 'P', 'CAT': 'H', 'CAC': 'H', 'CAA': 'Q', 'CAG': 'Q',
     'CGT': 'R', 'CGC': 'R', 'CGA': 'R', 'CGG': 'R', 'ATT': 'I',
     'ATC': 'I', 'ATA': 'I', 'ATG': 'M', 'ACT': 'T', 'ACC': 'T',
     'ACA': 'T', 'ACG': 'T', 'AAT': 'N', 'AAC': 'N', 'AAA': 'K',
     'AAG': 'K', 'AGT': 'S', 'AGC': 'S', 'AGA': 'R', 'AGG': 'R',
     'GTT': 'V', 'GTC': 'V', 'GTA': 'V', 'GTG': 'V', 'GCT': 'A',
     'GCC': 'A', 'GCA': 'A', 'GCG': 'A', 'GAT': 'D', 'GAC': 'D',
     'GAA': 'E', 'GAG': 'E', 'GGT': 'G', 'GGC': 'G', 'GGA': 'G',
     'GGG': 'G', }

# Extra data in case you want it.
stop_codons = [ 'TAA', 'TAG', 'TGA']
start_codons = [ 'TTG', 'CTG', 'ATG']
```