COMP 204: Sequence alignment examples, more dictionaries

Yue Li
based on material from Mathieu Blanchette, Carlos Oliver, Christopher J.F. Cameron
Midterm materials coverage and practice midterms

- Midterm is held on February 22 at 6:30-8:00 pm in LEA 219.
- Our midterm will cover up to Lecture 17 (Feb 13)
- Past midterms in COMP 204 Fall 2018 and COMP 364 Fall 2017 are posted on myCourses for practice
The most important of Assignment 2 is to understand Needleman-Wunsch global sequence alignment algorithm. Let’s do a couple of examples together:

**Example 1**
Sequence 1: G
Sequence 2: GCG

**Example 2**
Sequence 1: TCGA
Sequence 2: TTCG
A matrix in Python is just a list of lists of the same length

In Assignment #2, we will need to represent two-dimensional tables or matrices, with a fixed number of rows and columns. *Two-dimensional lists* can be used to do this in Python. A 2D list is a list of lists, where each of the lists is of the same length. Example: A tic-tac-toe grid:

```python
# A 2D list is a list of lists, where each of the lists is of the same length.
tictactoe = [ ['X', '', 'O'], [' ', 'X', ' '], ['O', '', ' ']

print(tictactoe)  # [['X', '', 'O'], [' ', 'X', ' '], ['O', '', ' ']]
```

To access an element in a 2D list, specify the index of the row and column:

```python
tictactoe[1][2] = 'X'
print(tictactoe)  # [['X', '', 'O'], [' ', 'X', 'X'], ['O', '', ' ']]
```

To create a new matrix with zeros we can use *list comprehension*:

```python
# Create an alignment scoring grid for two DNA sequences
seq1="GATTACA"
seq2="GCATGCA"
alignmentScoreGrid=[[0 for j in range(len(seq2)+1)]
    for i in range(len(seq1)+1)]
print(alignmentScoreGrid)  # print out the scoring grid
```
Dictionaries Recap

- A dictionary is said to be a **mapping** type because it maps **key** objects to **value** objects.
- Dictionaries are immensely useful and are the magic behind a lot of Python functionality.
- **Syntax:**  
  ```python
  my_dict = {[key]: [value], ...,}
  ```
- The analogy to a real dictionary works. The word you look up is the **key** and the definition is the **value**.
Dictionaries: picture

- Keys map to values.
- We use dictionaries when we want to access data using something other than an index (i.e. lists).
Dictionaries: keys and values

- A dictionary’s keys can be many different types of **immutable** objects (i.e. int, str, tuple)
- You can access a key’s value like a list. **Syntax:**
  ```python
  my_dict[key]
  ```
- You can mix and match key types
- Values can be any object type. You can also mix and match.

```python
record_sales = {
    "Kanye": 2.4,
    "Beyonce": 1.5,
    "Chance": 1.2,
    ("a", 12): "bob"
}
print(record_sales["Beyonce"] ) # 1.5
print(record_sales[ ("a", 12) ] ) # "bob"
```
Adding keys to a dictionary

Syntax: \[\text{my\_dict["key"] = value}\]

- If the key does not yet exist, a new key/value pair is created.
- If the key already exists, its previous value is overwritten.

```python
defining a dictionary:
>>> d = {"bob": 28}
>>> print(d)
{"bob": 1.2}

adding a new key:
>>> d["charlie"] = 33
>>> print(d)
{"bob": 1.2, "charlie": 2.5}

overwriting a key:
>>> d["bob"] = "woooo"
{"bob": "woooo", "charlie": 33}

deleting a key:
>>> del d["bob"]  # we can delete keys with the del operator
{"charlie": 33}
```
Dictionaries are **mutable** We can modify the contents of the dictionary as much as we want.

```python
>>> d = {"bob": 24, "tina": 11}
>>> d["tameeka"] = 42
>>> d['bob'] = [1, 2, 3, 4]
>>> del d["bob"]
>>> mystring = 'AAAGGG'
>>> mystring[2] = 'T' # this is an error. strings are immutable
```
Important properties of dictionaries

▶ Key-value pairs are **NOT** always stored in order. (for the current Python 3.7 they are, but assume it won’t be like this forever)

▶ If you want to iterate over the keys in a dictionary use the `dict.keys()` function.

```python
>>> d = {"bob": 24, "tina": 11}
>>> for k in d.keys():
...    print(k)
"tina"
"bob"
```
Useful dictionary methods and operators

- \texttt{d.items()} produces an iterator which yields tuples of the form \((\text{key}, \text{value})\)

```python
>>> for k,v in d.items():
...     print("key:", k, "value:", v)

key: bob, value: 24
key: tina, value: 11
```

- \texttt{k in d} evaluates to \texttt{True} if the key exists in the dictionary and \texttt{False} otherwise.

- \texttt{d.update(d2)} “merges” two dictionaries into one.

```python
>>> d = \{"a": 3, "b": 4\}
>>> d.update(\{"c": 5\})
>>> d
\{"a": 3, "c": 5, "b": 4\}
```
BLAST is a very popular bioinformatics tool used to compare DNA sequences. One of the main innovations is to index a genome by 'words'.

words are short sequences. AT, CG, CC, GG, AA

Goal: Given a genome and a list of words return a dictionary with a list of positions where each given word occurs.

Example: for words AAG, AAT in genome GAAGAAGGGAATGGAAGAAT we should return ‘AAG’: [1, 4, 14], ‘AAT’: [9, 17].

Note BLAST is a heuristic approach to do fast sequence search but Needleman-Wunsch global alignment algorithm (or Smith-Waterman local alignment) is a more principled way to find optimal match(es) at the cost of speed.
Building genomic dictionary

```python
#Args: genome_seq: a DNA sequence as a string
# words: an iterable of sequences
#Returns:
# genomeDict: a dict with a key for each word mapping to
# list of indices.
def buildGenomeDict(genome_seq, words):
    genomeDict = {}
    for w in words:
        for i in range(len(genome_seq)-len(w)+1):
            if genome_seq[i:i+len(w)] == w:
                if w not in genomeDict:
                    genomeDict[w] = []
                genomeDict[w].append(i)
    return genomeDict

genome_seq = "AGCGACGTATAATCGACTA"
words=["CG", "TATA"]
genomeDict = buildGenomeDict(genome_seq, words)
print(genomeDict)
# {'CG': [2, 13], 'TATA': [7]}
```
Searching genomic dictionary

#Args: genomeDict: build from genome_index
#    genome_seq: DNA sequence corresponding to genomeDict
#    queries: a list of query sequences
#Returns: blasthits: a dict with a key for each query and their genomic location(s)
def searchGenomeDict(genomeDict, genome_seq, queries):
    blasthits = {}
    for q in queries:
        blasthits[q] = []  # initialize query hit list
        for genomeDictKey in genomeDict.keys():
            for i in range(len(q) - len(genomeDictKey) + 1):
                wordlen = len(genomeDictKey)
                querySubstr = q[i:i+wordlen]
                if querySubstr == genomeDictKey:
                    genomePosList = genomeDict[querySubstr]
                    for pos in genomePosList:
                        if genome_seq[pos-i:pos+len(q)-i] == q:
                            blasthits[q].append(pos-i)
                if len(blasthits[q]) > 0:
                    blasthits[q] = set(blasthits[q])  # set returns unique values (more info on set in the next lecture)
    return blasthits

queries = ["ACGT", "CGACGT", "TATAAT", "CGACT", "XYZ"]
myhits = searchGenomeDict(genomeDict, genome_seq, queries)
print(myhits)  # {'ACGT': {4}, 'CGACGT': {2}, 'TATAAT': {7},...}
A convenient method: setdefault

Let’s look at line 15 in the previous example:

```python
if w not in word_index:
    word_index[w] = []
word_index[w].append(i)
```

You will find yourself writing this statement many times.

```python
mydict.setdefault(key, [default])
```

If key is in the dictionary, return its value. If not, insert key with a value of default and return default.

We can replace it with one line using `setdefault`

```python
word_index.setdefault(w, []).append(i)
```
Dictionaries Pop Quiz

True or False: dictionaries are immutable.

Error? myd = {[1,2]: "hello"}

True or False: dictionary keys must be unique.

Error? d2 = {'bob': 2, 'susan': 3, 'bob':4}

Error? d = {}; d['bob'].append(3)

True/False: once a key-value is stored we can’t update it.
Dictionaries Pop Quiz Answers

- True or False: dictionaries are immutable.
  - Error? `myd = {[1,2]}: "hello"` – Yes. Keys must be immutable.

- True or False: dictionary keys must be unique.
  - Error? `d2 = {'bob': 2, 'susan': 3, 'bob':4}` – No. Duplicate keys are overwritten.

- Error? `d = {}; d['bob'].append(3)` – Yes. Key 'bob' has not been created.

- True/False: once a key-value is stored we can’t update it. `d['bob'] = 3; d['bob']='hi'` is valid.