CSE 503

Introduction to Computer Science for Non-Majors

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Day 13
Dictionaries

Recap

- Mistakes will happen and things will go wrong
- How do we fix it? First we need to figure out what is going wrong.
  - We can use output (print or console.log) to give more information about what is happening during execution.
  - Checking that the output matches with our expectations can help reveal where things have gone wrong.
  - Asserts can also be used to automate the checking, and error messages can give more details on why a failure occurs.

DNA Frequency Exercise from Last Time

Write a function called **`dnaFrequency`** that takes a single DNA string, and returns a list of 4 lists, one for each base and its count.

For example:

`dnaFrequency("ACAGCCTAAG")` must return

```
[["A", 4],["C", 3], ["G",2],["T",1]]
```

`dnaFrequency("TCAGCCTAAG")` must return

```
[["A", 3],["C", 3], ["G",2],["T",2]]
```

DNA Frequency

```
1.  def dnaFrequency(string):
2.    bases = "ACGT"
3.    l = []
4.    for b in bases:
5.      l.append([b, dnaCount(string, b)])
6.
7.    return l
```

Was there anything…odd about our solution?

-

Was there anything...odd about our solution?

- What order do we put the lists in? Why?
  - Now we have to remember this order...

Was there anything...odd about our solution?

- What order do we put the lists in? Why?
  - Now we have to remember this order...
- How would we access the count for "A", for example?
  - `f = dnaFrequency("AACTACGGCT")`
  - `f[0][1]`
  - That is awkward. What ties that to "A"?
  - What if the order changes?

Was there anything...odd about our solution?

- What order do we put the lists in? Why?
  - Now we have to remember this order...
- How would we access the count for "A", for example?
  - `f = dnaFrequency("AACTACGGCT")`
  - `f[0][1]`
  - That is awkward. What ties that to "A"?
  - What if the order changes?
- How would we prefer to access the data?

Ordered vs Associative

- So far the collections we've seen (lists and arrays) have been ordered
  - They store a collection of values in a specific order
  - We access elements by their position in the list
    - ie a[0], a[3], a[147]
- Associative collections are different type of collection we can use in both Python and JavaScript
  - These collections associate a *key* with a *value* (called a <key, value> pair)
  - We access elements by their key

Key-Value Pairs in Real Life

<"First Name":"Eric">

<"Occupation":"Lecturer">

<"Siblings":3>

<"UBIT":"epmikida">

<"Last Name":"Mikida">

<"Favorite Number":2>

Key-Value Pairs in DNA Example

<"A": 17>

<"C": 4>

<"T": 9>

<"G": 14>

Python: Dictionary

- In Python, a key-value mapping is called a Dictionary
  - Dictionaries are indexed by *key* (instead of by a position)
  - A dictionary consists of a collection of key:value pairs, with the requirement that keys are unique
  - Strings can be keys, but so can any other value

Python Dictionary

The delimiters used to specific dictionaries are curly braces { }

Python Dictionary

The delimiters used to specific dictionaries are curly braces `{ }`

An empty dictionary can be created with a set of braces:

```
d1 = {}
```

Python Dictionary: Creation

The delimiters used to specific dictionaries are curly braces `{ }`

An empty dictionary can be created with a set of braces:

```
d1 = {}
```

A dictionary can be given initial key:value pairs by giving it a comma separated list of key:value pairs inside the braces. This is also how dictionaries are printed as output.

```
d2 = {'A':6, 'C':3, 'G':1, 'T':2}
```

Python Dictionary: Element Access

Square brackets can be used to add/update/access individual items:

```
d = {"name":"Eric"}
d["age"] = 32 # Brackets can add a key:value pair
d["age"] = 29 # They can also update an existing pair
print(d["age"]) # ...or just to access a value
```

Python Dictionary: Element Access

Square brackets can be used to add/update/access individual items:

```
d = {"name":"Eric"}
d["age"] = 32 # Brackets can add a key:value pair
d["age"] = 29 # They can also update an existing pair
print(d["age"]) # ...or just to access a value
```

The **`update`** function can be used to add/update from another dictionary

```
d.update({"age":50, "job":"Lecturer"})
```

Python Dictionary: Element Access

The **get** function provides a different way to access values

```
# Behave the same if the key exists
print(d["name"])     # Prints "Eric"
print(d.get("name")) # Prints "Eric"
```

Python Dictionary: Element Access

The **get** function provides a different way to access values

```
# Behave the same if the key exists
print(d["name"])     # Prints "Eric"
print(d.get("name")) # Prints "Eric"

# Behave different when the key does not exist
print(d["salary"])              # Error! Key not in dictionary
print(d.get("salary"))          # No error, no return value
print(d.get("salary", False)) # Returns false
```

Python Dictionary: Removal

Items can be removed with the **`del`** keyword, or **`pop`** function

```
del d["age"] # Removes "age", returns nothing
d.pop("job") # Removes "job", returns its value
print(d)     # Now d is just {"name":"Eric"}
```

Membership can be tested with **`in`** and **`not in`**

```
"name" in d     # Would evaluate to True
"age" in d      # Would evaluate to False (age was just removed)
"job" not in d # Would evaluate to True
```

Python Dictionary: Keys, Values, Items

Dictionaries provide access to *sequences* for keys, values, and pairs

```
d = {"Manager":"Sally", "Cashier":"Bob", "Security":"Joel"}
for k in d.keys():   # Will print out "Manager", "Cashier", etc...
  print(k)

for v in d.values(): # Will print out "Sally", "Bob", "Joel"
  print(v)

for x in d.items():  # Will print out ("Manager", "Sally"), etc...
  print(x)
```

DNA Frequency Revisited

Write a function called **`dnaFrequency`** that takes a single DNA string, and returns a *dictionary* containing the frequency of each base.

For example:

`dnaFrequency("ACAGCCTAAG")` must return
`{"A":4,"C":3,"G":2,"T":1}`

How does this compare to the list version?