COMP 364: Dictionaries, Sets, and Commenting

Carlos G. Oliver, Christopher J.F. Cameron

October 19, 2017

Outline

1. Recap
2. Dictionaries & Sets
3. Commenting & Coding Style
4. Practice Problem

Exception Handling

```
1  try:
2      #try this code
3  except ExceptionName:
4      #if ExceptionName is raised by try block do this
5  except OtherExceptionName:
6      #if OtherExceptionName is raised do this
7  else:
8      #if no exception is raised by try do this
9  finally:
10     #this ALWAYS executes right after the try
```

Warm-up: List comprehensions

- Write a function that takes a list of integers as input and returns a list containing only the even numbers in the original list.
- Do this in three ways:
  1. Using a while loop
  2. Using a for loop
  3. Using a list comprehension

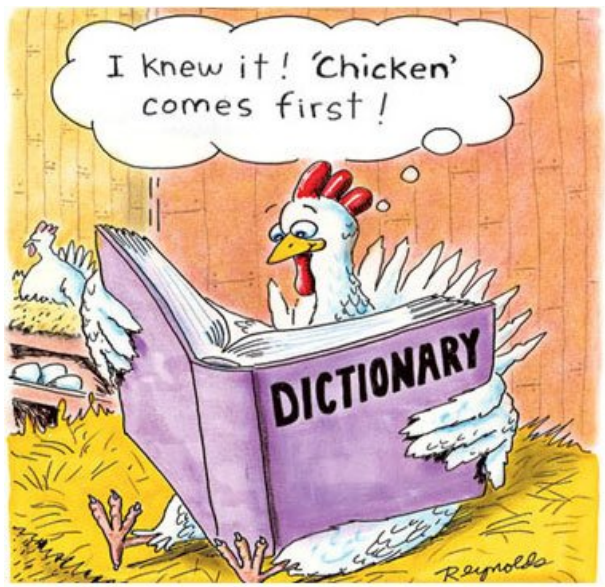
Life Hack 1

If you don't plan on using the loop variable you don't have to give it a name, just put an underscore instead.

```
>>> rihanna = ["work " for _ in range(5)]
["work", "work", "work", "work", "work"]
```

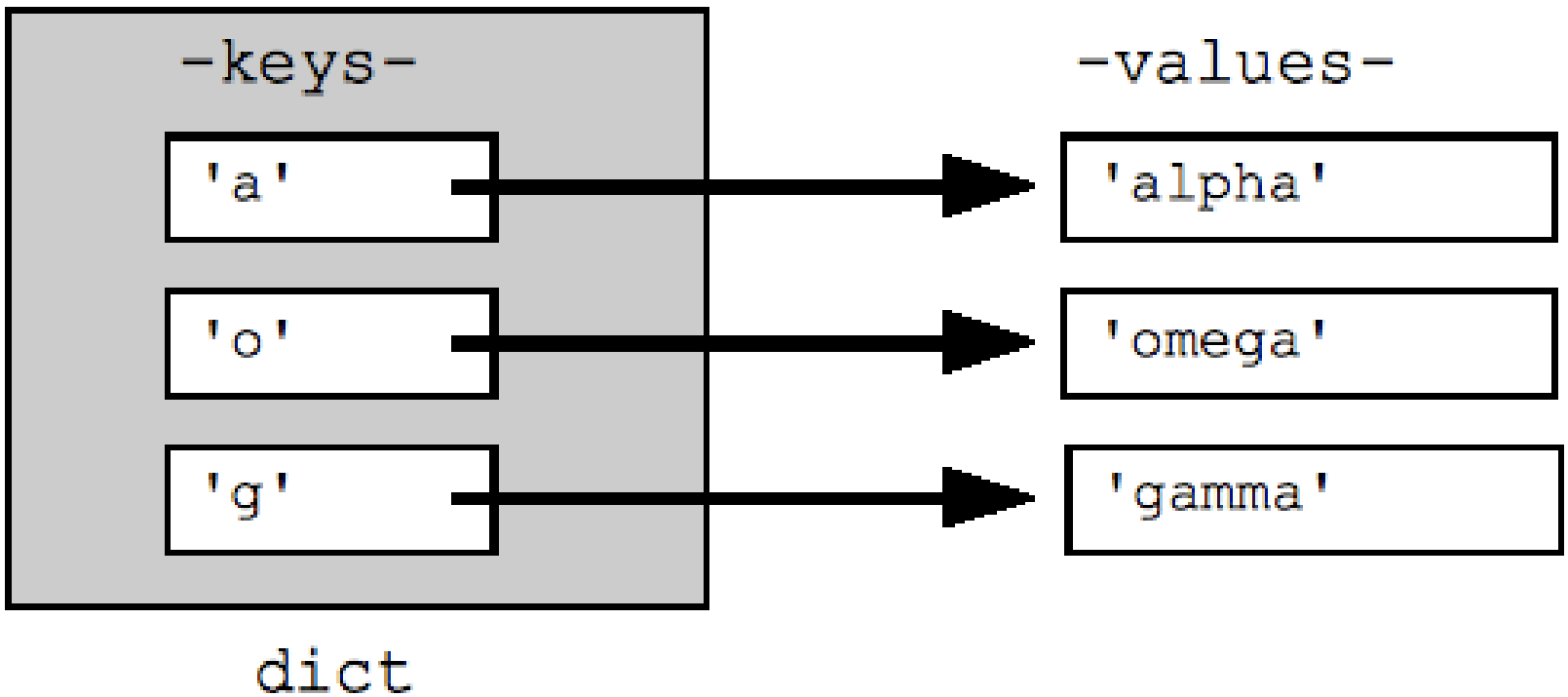
A very useful type: Dictionary

- 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: `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.



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: `my_dict[key]`
- You can mix and match key types
- Values can be any object type. You can also mix and match.

```
record_sales = {
    "Kanye": 2.4,
    "Kendrick": 1.5,
    "Chance": 1.6,
    12: lambda x: x / 2,
    ("a", 12):  "bob"
    }
print(record_sales["Kendrick"]) #1.5
print(record_sales[("a", 12]) #2
print(record_sales[12](10) # 5
```

Adding keys to a dictionary

- Syntax: `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

```
>>> d = {"bob": 28}
>>> print(d)
{"bob": 1.2}
>>> d["charlie"] = 33
>>> print(d)
{"bob": 1.2, "charlie": 2.5}
>>> d["bob"] = "woooo"
{"bob": "woooo", "charlie": 33}
```

Important properties of dictionaries

- Dictionaries are **mutable**

```
>>> d = {"bob": 24, "tina": 11}
>>> id(d)
4299176768
>>> d["tameeka"] = 42
>>> id(d)
4299176768
>>> del d["bob"]
```

Important properties of dictionaries

- Key-value pairs are **NOT** always stored in order. (for the current Python 3.6 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.

```
>>> d = {"bob": 24, "tina": 11}
>>> for k in d.keys():
>>> ... print(k)
"tina"
"bob"
```

Useful dictionary methods and operators

- `d.items()` produces an iterator which yields tuples of the form `(key, value)`

```
>>> [f"key: {k}, value: {v}" for k, v in
↪  d.items()]
["key: bob, value: 24, "key: tina, value: 11]
```

- `k in d` evaluates to `True` if the key exists in the dictionary and `False` otherwise.
- `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`.
- `d.update(d2` "merges" two dictionaries into one.

```
>>> d = {"a": 3, "b": 4}
>>> d.update({"c": 5}
{"a": 3, "c": 5, "b": 4}
```

Quick dictionary example: Counting nucleotides

```
1  sequence = "AAAGGGAAGUGUAUGUGAAACCCC"
2  seq_counts = {}
3  for s in sequence:
4      if s not in seq_counts:
5          seq_counts[s] = 1
6      else:
7          seq_counts[s] += 1
8  ## easier: use setdefault
9  for s in sequence:
10     seq_counts[s] = seq_counts.setdefault(s, 0) + 1
```

Sets: the unordered container for unique things

- Syntax: `myset = {1, 2, 3}` or `myset = set([1, 2, 3])` (careful, `myset = {}` is an empty dictionary)
- Sets never contain duplicates. Python checks this using the `==` operator.

```
1  >>> myset = set([1, 1, 2, 3])
2  set([1,2 , 3]) #only keep unique values
3  >>> myset.add(4)
4  set([1, 2, 3, 4])
5  >>> myset.add(1)
6  set([1, 2, 3, 4])
7  #get unique characters of string
8  >>> charset = set("AAACCGGGA")
9  {A, C, G}
```

- Sets can only contain immutable objects (like dictionary keys)
- Elements in sets do not preserve their order.

Useful set methods and operations

- Membership testing

```
>>> 4 in myset
False
```

- Set intersection (elements common to A and B, if A and B are sets)

```
>>> A = {"a", "b", "c"}
>>> B = {"a", "b", "d"}
>>> A & B # equivalent to: A.intersection(B)
set(["a", "b"])
```

- Click here for a full list of set functionality.

Useful set methods and operations

- Set difference (elements in A that are **not** in B)

```
>>> A - B
set(["c"]) #same as: A.difference(B)
```

- Set union (Elements found in A or B)

```
>>> A | B # equivalent to: A.union(B)
set(["a", "b", "c", "d"])
```

- These can be applied to multiple sets

```
>>> C = {"a", "c", "d", "e"}
>>> A & B & C # A.intersection(B, C)
set(["a"]) #elements common to A and all others
```

The `Collections` module: `import collections`

- Implementations of standard data types made easier.
- `namedtuple()` a tuple where indices can be given names
- `Counter` dictionary that counts objects for you

```
>>> cnt = collections.Counter(['c', 'b', 'b',
↪  'c', 'a', 'c'])
Counter({'b': 2, 'a': 1, 'c':3})
>>> cnt.most_common(2) # 2 most common items
[('c', 3), ('b', 2)]
```

- Try the nucleotide counting example from above using Counter.
- `OrderedDict` dictionary that keeps the order entries as added.

The `Collections` module: `import collections`

- Implementations of standard data types made easier.
- `namedtuple()` a tuple where indices can be given names

```
1  >>> Grade = collections.namedtuple('Grade',
   ↪  ['name', 'gpa']) #define the named tuple
2  >>> mygrade = Grade("Carlos", 3.1)
3  >>> mygrade[0]
4  "Carlos"
5  >>> mygrade.gpa
6  3.1
```

- `OrderedDict` dictionary that keeps the order entries as added. (more info)

Commenting: rules of thumb

- Comments should be informative but not overly detailed.
- Comments should be indented with the block they address
- When in doubt: it is always better to comment than to not comment

Which is better?

```
#this line binds an empty list to the name
 ↪  'students'
students = []
for s in students:
#loop over list and print
    print(s)
```

```
#keep track of students in a list
students = []
#display student list
for s in students:
    print(s)
```

Commenting: Docstrings

- A triple quoted string directly under a function header is stored as function documentation.

```
1  def my_max(lili):
2      """ Input: an iterable
3            return: max of list
4      """
5      return max(lili)
```

```
1  >>> help(my_max)
2  Help on function my_max in module __main__:
3
4          my_max(lili)
5                  Input: an iterable
6                  return: max of list
```

Tips on coding style

- Be critical of your code. → is this the best it can be?
- Avoid hard-coding
  - `for i in range(len(mylist))` is better than
  - `for i in range(5)`
- "Flat is better than nested" when you have too many loops inside loops you are probably doing something wrong.
- Give objects informative names
- When lines get too long you are either doing something wrong or you should break the line using forward slash.

```
1 for mylistitem in [innerlistitem in \
2     originallist if innerlistitem / 2 + 4 > 9]:
3     print("hi")
```

- A complete description of Python's coding style guidelines is here

Practice problem

Tinder has called you again to implement a "no pictures" mode.

- Register user
- Show potential candidates
- Check for match

Life Hack 2

The built in module `random` will come in handy.

```
1 import random
2 #random number
3 rand_num = random.random()
4 #random list item
5 rand_choice = random.choice(["bob", "alice",
  ↪  "mary"])
```