04-24-2023

# CS051A

# INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

## 23: Exceptions and sets


Alexandra Papoutsaki

she/her/hers

Lectures



Zilong Ye

he/him/his

Labs

# Lecture 23: Exceptions and sets

- Administrative
- Exceptions
- Sets

# Administrative

- Last assignment!
  - You will build a web crawler for Pomona webpages.
    - No Ethics reading
- Apply to be a TA for CS51P
- Apply to be a student liaison

# Lecture 23: Exceptions and sets

- Administrative
- Exceptions
- Sets

# `list_max` function in `exceptions.py`

- Are there any list inputs that would give an error?
  - non-numerical
  - empty lists
- How could we fix this?
  - check if its equal to the empty list
  - print an error message
  - return ???
- A better way to fix this is to raise an exception (like you've probably seen for other problems)

# Exceptions

- Exceptions are another way of communicating information from a function/expression:

```
>>> 1/0
Traceback (most recent call last):
 Python Shell, prompt 3, line 1
builtins.ZeroDivisionError: division by zero
```

- they allow us to give information back from a function besides return.
  - if we don't do anything about them, exceptions will cause the program to terminate.

# Raising exceptions

- Look at the `list_max_better` function in `exceptions.py`
  - to raise an exception, you use the keyword `raise` and then create a new `Exception` object

```
>>> list_max_better([1, 2, 3])
3
>>> list_max_better([])
Traceback (most recent call last):
Python Shell, prompt 3, line 1
# Used internally for debug sandbox under external interpreter
File "/Users/apaa2017/classes/cs51a/examples/exceptions.py", line 12,
in <module>
raise Exception("list must be non-empty")
builtins.Exception: list must be non-empty
```

# `get_scores` function in `exceptions.py`

- Are there any inputs that the user could enter that would cause a problem? Specifically, cause the function to exit early?

```
>>> get_scores()
Enter the scores one at a time. Blank score finishes.
Enter score: 1
Enter score: banana
Traceback (most recent call last):
Python Shell, prompt 2, line 1
#Used internally for debug sandbox under external interpreter
File "/Users/apaa2017/classes/cs51a/examples/exceptions.py", line 29,
in <module>
scores.append(float(line))
builtins.ValueError: could not convert string to float: 'banana'
```

# `get_scores` function in `exceptions.py`

- If we enter a non-numerical value, we get a “ValueError”.
- What would you like to do instead?
  - It’s better to prompt the user to enter a number and try again
    - How can we do this?
      - One way would be to check that the string is a valid number.
        - kind of a pain (decimal numbers, positive/negative numbers, even scientific notation is fair game, e.g., 1.3e10).
- Better way: handle the exception and deal with it.

# try/except

- We can catch an exception and deal with it using a try/handle block:

```
try:
        # Some code that could raise an exception
except ExceptionName:
        # what to do if exception occurs
```

- The code in the block is executed.
- If no exception is raised, the code finishes and the code in the except block is skipped and the code keeps running.
- If an exception occurs, the code in the try block is immediately exited.
  - If it's of the type in the except block, the code in the except block executes and then the code keeps running after that.
  - if it's another exception, it exits.

## `get_scores_better` function in `exceptions.py`

- We can handle the `ValueError` exception and print out an error message, but keep going

```
>>> get_scores_better()
Enter the scores one at a time. Blank score
finishes.
Enter score: 1
Enter score: banana
Enter numbers only!
Enter score: 2
Enter score:
[1.0, 2.0]
```

## `print_file_stats` function in `exceptions.py`

- where could we get exceptions from this code?
  - file doesn't exist!
  - if the file is empty, then we could also get a divide by zero error.

# `print_file_stats_better` function in `exceptions.py`

- if we have multiple exceptions, we can have multiple except blocks.
- Each block will only be executed if an exception of that type is raised.
- In the case of the divide by zero error, we'll already have printed out some information (number of words, longest word, shortest word). All we want to do is not have an error raised.

## `print_file_stats_better` function in `exceptions.py`

- `pass`
  - certain control statements expect code to be there (e.g., if/then, try/except).
- pass can be used as a non-operation: it is code, but it doesn't do anything.

# Lecture 23: Exceptions and sets

- Administrative
- Exceptions
- Sets

# Sets

- what is a set, e.g., a set of data points?
  - an unordered collection of data
- How does this differ from a list?
  - a list has a sequential order to it
- What operations/methods might we want from a set?
  - create new/construct a set
  - add things to the set
  - remove things from the set
  - ask if something belongs in the set
  - intersection of two sets
  - union of two sets

# set class

- >>> help(set)
- We can construct a new set using a constructor or using {} (kind of like dictionaries).

```
>>> s = set()
>>> s
{}
>>> s = set([4, 3, 2, 1])
>>> s
{1, 2, 3, 4}
>>> s = {4, 3, 2 ,1}
{1, 2, 3, 4}
>>> s = set("abcd")
>>> s
{'a', 'c', 'b', 'd'}
>>> s = {1, 1, 1, 1, 2, 2}
{1, 2}
```

- notice that there are two constructors, the empty constructor (set()), which created an empty set and a constructor that took a single parameter, anything that is iterable, e.g., a list, a string, in general, any thing that we can iterate over in a for loop.
- Notice that even though we may give it something where there is ordering, the ordering is NOT guaranteed to be preserved.

# `set` methods

- From the help output, which of the following are mutator vs. accessor?
  - `add`
  - `clear`
  - `difference`
  - `difference_update`
  - `intersection`
  - `intersection_update`, …
- Other interesting methods:
  - `pop`
  - `remove`
  - `isdisjoint`
  - `issubset`
  - `issuperset`
  - `union`
  - `update`

# set methods

- supports most of the methods you'd want for a set:

```
>>> s = {1,2,3,4}
>>> s.add(5)
>>> s
{1, 2, 3, 4, 5}
>>> s2 = set([4, 5, 6, 7])
>>> s2
{4, 5, 6, 7}
>>> s.difference(s2)
{1, 2, 3}
>>> s
{1, 2, 3, 4, 5}
>>> s2
{4, 5, 6, 7}
>>> s.union(s2)
{1, 2, 3, 4, 5, 6, 7}
>>> s.intersection(s2)
{4, 5}
>>> s
{1, 2, 3, 4, 5]}
>>> s2
{4, 5, 6, 7}
```

# set methods

- we can also ask if an item is in a set:

```
>>> s = {1,2,3,4}
>>> s2 = set([4, 5, 6, 7])
>>> 1 in s2
False
>>> 5 in s2
True
>>> "abc" in s2
False
>>> s2 in s2
False
```

- Notice that you CANNOT index into a set (there is no order)

```
>>> s[0]
Traceback (most recent call last):
File "<string>", line 1, in <fragment>
TypeError: 'set' object does not support indexing
```

# Why sets?

- seems like we could do all of these things and more with lists?
  - `list` has all of the operations like *add*, *pop*, `find` that sets have.
  - Sets have some nice operations like `union` and `intersection`, but we could put these in the `list` class
  - In fact, lists also support the "in" notation

```
>>> some_list = [1, 2, 3, 4]
>>> 4 in some_list
 True
>>> "abc" in some_list
False
```

- Why have the separate class for set then?
  - Performance!

# Why sets?

- Write the following function:
  - contains(list, item)
    - returns True if the item is in the list
    - false otherwise
    - don't use "in" or "find"
- def contains(list, item):

```
    for thing in list:
            if thing == item:
                    return True
        return False
```

- If we're searching for an item and we double the size of the list, how much longer (on average) do you think it would take to run this function?
  - Twice as long since we're looping through each item in the list
    - Computers are fast, but there still is a cost to each operation
  - What if we quadrupled the size of the list?f
    - Four times as long
- The `contains` function above is called a "linear" runtime function

# lists_vs_sets.py

- Two functions for generating data:
  - `generate_set`: generates random points and puts them into a set.
  - `generate_list`: generates random points and puts them into a list.
- `query_data`
  - generates `num_queries` random numbers
  - uses `in` to see if they are in the data set
  - times how long it takes to do num_queries
- `speed_test`
  - generates equal sized data sets in both list and set form
  - then calls `query_data` to see how long it takes to query each one

# lists_vs_sets.py

- ```
  >>> speed_test(1000, 100)
  List creation took 0.003422
  seconds
  Set creation took 0.003589
  seconds
  --
  List querying took 0.002917
  seconds
  Set querying took 0.000194
  seconds
  ```

- For small sizes, they behave fairly similarly. As we increase the size of the set and the number of queries, however, we start to see some differences

```
>>> speed_test(10000, 100)
List creation took 0.023313 seconds
Set creation took 0.021885 seconds
--
List querying took 0.021288 seconds
Set querying took 0.000179 seconds
```

```
>>> speed_test(10000, 1000)
List creation took 0.020332 seconds
Set creation took 0.021198 seconds
--
List querying took 0.213577 seconds
Set querying took 0.001833 seconds
```

```
>>> speed_test(100000, 1000)
List creation took 0.186876 seconds
```
```

# lists_vs_sets.py

- we can better understand these by generating points as we increase the size of the set/list and then plotting them (we can do it in Excel or in Python)

```
>>> speed_data(5000, 10000, 100000, 5000)
size   list   set
10000   0.237790   0.001881
15000   0.358325   0.001999
20000   0.469743   0.001956
25000   0.602107   0.001916
30000   0.687776   0.001889
35000   0.824027   0.001903
40000   0.921235   0.001952
45000   1.009843   0.001912
50000   1.156059   0.001927
55000   1.386080   0.001913
60000   1.566058   0.001984
```

```
60000   1.566058   0.001984
65000   1.722870   0.001936
70000   2.025138   0.001966
75000   2.363384   0.001962
80000   2.619580   0.002030
85000   2.897005   0.002054
90000   2.975576   0.001946
95000   3.418256   0.002082
```

# When to use a set vs a list?

- Lists have an ordering.
- If you need indexing, use a list
- Sets are faster for asking membership
- if you don't care about the order, use a set!

## Resources

- Textbook: Chapter 3 and Chapter 13
- `exceptions.py`
- `lists_vs_sets.py`

## Homework

- Assignment 12