

DICTIONARIES, DEBUGGING, EXCEPTIONS

(download slides and .py files from Stellar to follow along)

6.0001 LECTURE 6

Eric Grimson

6.0001 LECTURE 6

1

TODAY

- dictionaries – another **mutable** object type
- debugging
- exceptions
- assertions

6.0001 LECTURE 6

3

DICTIONARIES

6.0001 LECTURE 6

5

LAST TIME

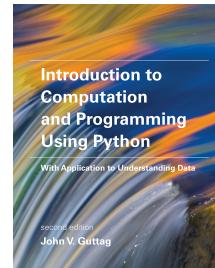
-
- tuples – immutable object type
 - lists – mutable object type
 - aliasing, cloning
 - mutability side effects
 - recursion

6.0001 LECTURE 6

2

Assigned Reading

- Section 5.6
- Chapter 6
- Chapter 7



https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

6.0001 LECTURE 6

4

HOW TO STORE STUDENT INFO



- could store using separate lists for each kind of information
- ```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.0, 6.0001, 20.002, 9.01]
```
- 2.0
- 6.0001
- 20.002
- 9.01
- a **separate list** for each item
  - each list must have the **same length**
  - info stored across lists at **same index**, each index refers to information for a different person
  - indirectly access information by finding location in lists corresponding to a person, then extract

6.0001 LECTURE 6

6

## HOW TO ACCESS STUDENT INFO



```
def get_grade(student, name_list, grade_list, course_list):
 i = name_list.index(student)
 grade = grade_list[i]
 course = course_list[i]
 return (course, grade)
```

**messy** if have a lot of different info of which to keep track  
 must maintain **many lists** and pass them as arguments  
 must **always index** using integers  
 could store each entry as [name, [course, grade]], but still need to linearly search list by name to find entry

6.0001 LECTURE 6

7

find location in list for person  
 Use location to access other info  
 Remember the “.” notation for methods associated with types of objects

- empty dictionary
- key
- value

## A BETTER AND CLEANER WAY – A DICTIONARY

nicer to **index item of interest directly** (not always int)

nice to use **one data structure**, no separate lists

| A list |        |
|--------|--------|
| 0      | Elem 1 |
| 1      | Elem 2 |
| 2      | Elem 3 |
| 3      | Elem 4 |
| ...    | ...    |

index element

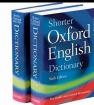
| A dictionary |      |
|--------------|------|
| Key 1        | Val1 |
| Key 2        | Val2 |
| Key 3        | Val3 |
| Key 4        | Val4 |
| ...          | ...  |

custom index by label element

6.0001 LECTURE 6

8

## A PYTHON DICTIONARY



- store pairs of data
- key
- value

|          |      |
|----------|------|
| 'Ana'    | 'B'  |
| 'Denise' | 'A'  |
| 'John'   | 'A+' |
| 'Katy'   | 'A'  |

empty dictionary  
 custom index by label  
 Note: values could be arbitrary structure

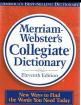
```
my_dict = {}
grades = {'Ana': 'B', 'Denise': 'A', 'John': 'A+', 'Katy': 'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

6.0001 LECTURE 6

9

## DICTIONARY LOOKUP



- similar to indexing into a list
- looks up the key**
- returns the value associated with the key**
- if key isn't found, get an error

|          |      |
|----------|------|
| 'Ana'    | 'B'  |
| 'Denise' | 'A'  |
| 'John'   | 'A+' |
| 'Katy'   | 'A'  |

```
grades = {'Ana': 'B', 'Denise': 'A', 'John': 'A+', 'Katy': 'A'}
grades['John'] → evaluates to 'A+'

grades['Sylvan'] → gives KeyError
```

6.0001 LECTURE 6

10

## DICTIONARY OPERATIONS

|          |      |
|----------|------|
| 'Ana'    | 'B'  |
| 'Denise' | 'A'  |
| 'John'   | 'A+' |
| 'Katy'   | 'A'  |
| 'Sylvan' | 'B'  |

```
grades = {'Ana': 'B', 'Denise': 'A', 'John': 'A+', 'Katy': 'A'}
add an entry
grades['Sylvan'] = 'C'
test if key in dictionary
'John' in grades → returns True
'Daniel' in grades → returns False
delete entry
del(grades['Ana'])
change entry
grades['Sylvan'] = 'B'
```

6.0001 LECTURE 6

11

## DICTIONARY OPERATIONS

|          |      |
|----------|------|
| 'Ana'    | 'B'  |
| 'Denise' | 'A'  |
| 'John'   | 'A+' |
| 'Katy'   | 'A'  |

```
grades = {'Ana': 'B', 'Denise': 'A', 'John': 'A+', 'Katy': 'A'}
get an iterable that acts like a tuple of all keys
grades.keys() → returns ['Denise', 'Katy', 'John', 'Ana']
get an iterable that acts like a tuple of all values
grades.values() → returns ['A', 'A', 'A+', 'B']

no guaranteed order
```

6.0001 LECTURE 6

12

## DICTIONARY KEYS & VALUES



- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - immutable** type (int, float, string, tuple, bool)
    - actually need an object that is **hashable**, but think of as immutable as all immutable types are hashable
  - careful with **float** type as a key
- no order** to keys or values!

```
d = {4:{1:0}, (1,3) :"twelve", 'const':[3.14, 2.7, 8.44]}
```

6.0001 LECTURE 6 13

| list                                                                                                                                                                                                          | vs | dict                                                                                                                                                                                                               |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li><b>ordered</b> sequence of elements</li> <li>look up elements by an integer index</li> <li>indices have an <b>order</b></li> <li>index is an <b>integer</b></li> </ul> |    | <ul style="list-style-type: none"> <li><b>matches</b> "keys" to "values"</li> <li>look up one item by another item</li> <li><b>no order</b> is guaranteed</li> <li>key can be any <b>immutable</b> type</li> </ul> |

6.0001 LECTURE 6 14

## EXAMPLE: THREE FUNCTIONS TO ANALYZE SONG LYRICS

- create a **frequency dictionary** mapping str:int
- find **word that occurs most often** and how many times
  - use a list, in case more than one word with same number
  - return a tuple (list, int) for (words\_list, highest\_freq)
- find the **words that occur at least X times**
  - let user choose "at least X times", so allow as parameter
  - return a list of tuples, each tuple is a (list, int) containing the list of words ordered by their frequency
  - IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dictionary.

6.0001 LECTURE 6 15

## CREATING A DICTIONARY



```
def generate_word_dict(song):
 song_words = song.lower()
 words_list = song_words.split()
 word_dict = {}
 for w in words_list:
 if w in word_dict:
 word_dict[w] += 1
 else:
 word_dict[w] = 1
 return word_dict
```

convert string into list of words  
can iterate over list of words in song  
if word in dict, increase #times you've seen it, update entry  
if word not in dict, seen word once, create entry

6.0001 LECTURE 6 16

## USING THE DICTIONARY



```
def find_frequent_word(word_dict):
 word = []
 highest = max(word_dict.values())
 for w in word_dict.keys():
 if word_dict[w] == highest:
 word.append(w)
 return (word, highest)
```

highest frequency in the dict's values  
create list of all words that have that freq

6.0001 LECTURE 6 17

## LEVERAGING DICT PROPERTIES



```
def occurs_often(word_dict, atleast):
 freq_list = []
 done = False
 while not done:
 word_freq_tuple = find_frequent_word(word_dict)
 if word_freq_tuple[1] < atleast:
 done = True
 else:
 freq_list.append(word_freq_tuple)
 for i in word_freq_tuple[0]:
 del(word_dict[i])
 return freq_list

song_dict = generate_word_dict(song)
print("***** WORDS IN SONG *****")
print(song_dict)
print("***** MOST COMMON WORD *****")
print(find_frequent_word(song_dict))
print("***** TOP MOST COMMON WORDS *****")
print(occurs_often(song_dict, 20))
```

Use Boolean as flag to stay/exit from a loop  
finding freq higher than atleast  
mutate dict

6.0001 LECTURE 6 18

## FIBONACCI RECURSIVE CODE

```
def fib(n):
 if n == 1:
 return 1
 elif n == 2:
 return 2
 else:
 return fib(n-1) + fib(n-2)
```

- two base cases
- calls itself twice
- this code is inefficient

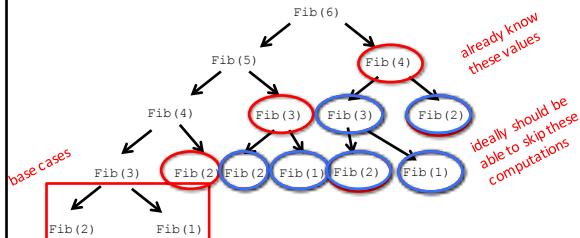


Leonardo Fibonacci

6.0001 LECTURE 6

19

## INEFFICIENT FIBONACCI

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$


- recalculating the same values many times!

- could keep track of already calculated values

6.0001 LECTURE 6

20

## FIBONACCI WITH A DICTIONARY

```
def fib_efficient(n, d):
 if n in d:
 return d[n]
 else:
 ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
 d[n] = ans
 return ans
```

*Method sometimes called "memoization"*

*Initialize dictionary with base cases*

- do a **lookup first** in case already calculated the value
- **modify dictionary** as progress through function calls

6.0001 LECTURE 6

21

## EFFICIENCY GAINS

- Calling  $\text{fib}(34)$  results in **11,405,773** recursive calls to the procedure
- Calling  $\text{fib\_efficient}(34)$  results in **65** recursive calls to the procedure
- Using dictionaries to capture intermediate results can be very efficient
- But note that this only works for procedures without side effects (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)



6.0001 LECTURE 6

22

## TESTING, DEBUGGING, EXCEPTIONS, ASSERTIONS

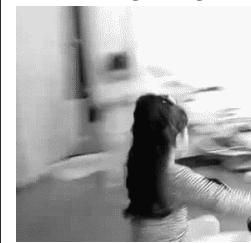


6.0001 LECTURE 6

23

## PROGRAMMING CHALLENGES

### EXPECTATION



### REALITY



What you want the program to do    What the program actually does

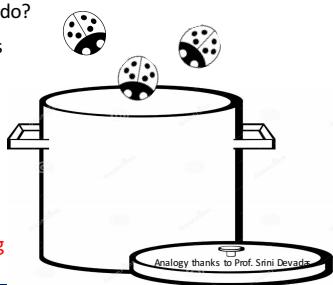
6.0001 LECTURE 6

24

## WE AIM FOR HIGH QUALITY – AN ANALOGY WITH SOUP

You are making soup but bugs keep falling in from the ceiling. What do you do?

- check soup for bugs
  - testing
- keep lid closed
  - defensive programming
- clean kitchen
  - eliminate source of bugs - debugging



Analogy thanks to Prof. Sriini Devadas

25

### DEFENSIVE PROGRAMMING

- Write **specifications** for functions
- **Modularize** programs
- Check **conditions** on inputs/outputs (assertions)

### TESTING/VALIDATION

- **Compare** input/output pairs to specification
- "Is it working?"
- "How can I break my program?"

### DEBUGGING

- **Study events** leading up to an error
  - "Why is it not working?"
  - "How can I fix my program?"

6.0001 LECTURE 6

26

## SET YOURSELF UP FOR EASY TESTING AND DEBUGGING

- from the **start**, design code to ease this part
- break program into **modules** that can be tested and debugged individually
- **document constraints** on modules
  - what do you expect the input to be? the output to be?
- **document assumptions** behind code design

"Motherhood and apple pie" approach:  
Something that cannot be questioned  
because it appeals to universally-held,  
wholesome values



6.0001 LECTURE 6

27

## WHEN ARE YOU READY TO TEST?

- ensure **code runs**
  - remove syntax errors
  - remove static semantic errors
  - Python interpreter can usually find these for you
- have a **set of expected results**
  - an input set
  - for each input, the expected output

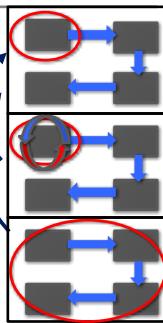


6.0001 LECTURE 6

28

## CLASSES OF TESTS

- **Unit testing**
  - validate each piece of program
  - **testing each function** separately
- **Regression testing**
  - fixing a bug might introduce new ones, so add test for bugs as you find them in a function
  - **catch reintroduced** errors that were previously fixed
- **Integration testing**
  - does **overall program** work?
  - most programmers tend to rush to do this



6.0001 LECTURE 6

29

## TESTING APPROACHES

- **intuition** about natural boundaries to the problem
 

```
def is_bigger(x, y):
 """ Assumes x and y are ints
 Returns True if y is less than x, else False """
 • can you come up with some natural partitions?
```
- if no natural partitions, might do **random testing**
  - probability that code is correct increases with more tests
  - better options below
- **black box testing**
  - explore paths **through specification**
- **glass box testing**
  - explore paths **through code**

6.0001 LECTURE 6

30

## BLACK BOX TESTING



```
def sqrt(x, eps):
 """ Assumes x, eps floats, x >= 0, eps > 0
 Returns res such that x-eps <= res*res <= x+eps """

```

- designed **without looking** at the code
- can be done by someone other than the implementer to avoid some implementer **biases**
- testing can be **reused** if implementation changes
- **paths** through specification
  - build test cases in different natural space partitions
  - also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

6.0001 LECTURE 6

31

## BLACK BOX TESTING



```
def sqrt(x, eps):
 """ Assumes x, eps floats, x >= 0, eps > 0
 Returns res such that x-eps <= res*res <= x+eps """

```

| CASE                   | x                  | eps                |
|------------------------|--------------------|--------------------|
| boundary               | 0                  | 0.0001             |
| Perfect square         | 25                 | 0.0001             |
| Less than 1            | 0.05               | 0.0001             |
| Irrational square root | 2                  | 0.0001             |
| extremes               | 2                  | $1.0/2.0^{**64.0}$ |
| extremes               | $1.0/2.0^{**64.0}$ | $1.0/2.0^{**64.0}$ |
| extremes               | $2.0^{**64.0}$     | $1.0/2.0^{**64.0}$ |
| extremes               | $1.0/2.0^{**64.0}$ | $2.0^{**64.0}$     |
| extremes               | $2.0^{**64.0}$     | $2.0^{**64.0}$     |

6.0001 LECTURE 6

32

## GLASS BOX TESTING



- **use code** directly to guide design of test cases
- called **path-complete** if every potential path through code is tested at least once
- what are some **drawbacks** of this type of testing?
  - can go through loops arbitrarily many times
  - missing paths
- guidelines
  - branches → exercise all parts of a conditional
  - for loops → loop not entered, body of loop executed exactly once, body of loop executed more than once
  - while loops → same as for loops, cases that catch all ways to exit loop

6.0001 LECTURE 6

33

## GLASS BOX TESTING



```
def abs(x):
 """ Assumes x is an int
 Returns x if x>=0 and -x otherwise """
 if x < -1:
 return -x
 else:
 return x
```

- a path-complete test suite could **miss a bug**
- path-complete test suite: 2 and -2
- but  $\text{abs}(-1)$  incorrectly returns -1
- should still test boundary cases

6.0001 LECTURE 6

34

## BUGS



- once you have discovered that your code does not run properly, you want to:
  - isolate the bug(s)
  - eradicate the bug(s)
  - retest until code runs correctly for all cases

6.0001 LECTURE 6

35

## DEBUGGING

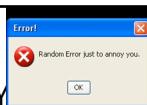


- steep learning curve
- goal is to have a bug-free program
- tools
  - **built in** to IDLE and Anaconda
  - **Python Tutor**
  - **print** statement
- use your brain, be **systematic** in your hunt

6.0001 LECTURE 6

41

## ERROR MESSAGES - EASY



- trying to access beyond the limits of a list  
test = [1, 2, 3] then test[4] → IndexError
- trying to convert an inappropriate type  
int(test) → TypeError
- referencing a non-existent variable  
a → NameError
- mixing data types without appropriate coercion  
'3' / 4 → TypeError
- forgetting to close parenthesis, quotation, etc.  
a = len([1, 2, 3])  
print a → SyntaxError

6.0001 LECTURE 6 42

## LOGIC ERRORS - HARD



- think before writing new code
- draw pictures
- take a break
- explain the code to
  - someone else
  - a rubber ducky

6.0001 LECTURE 6 43

## DEBUGGING STEPS

be a scientist



- study program code
  - ask how did I get the unexpected result
  - don't ask what is wrong
  - is it part of a family?
- scientific method
  - study available data – both correct test cases and incorrect ones
  - form an hypothesis consistent with the data
  - design and run a repeatable experiment with potential to refute the hypothesis
  - keep record of experiments performed: use narrow range of hypotheses, use simple test cases

6.0001 LECTURE 6 45

## PRINT STATEMENTS



- good way to test hypothesis
- when to print
  - enter function
  - show values of parameters before computation
  - function results
  - show value of computation before exiting

6.0001 LECTURE 6 47

## DEBUGGING AS SEARCH



- want to narrow down space of possible sources of error
- design experiments that expose intermediate stages of computation (use print statements!), and use results to further narrow search
- bisection search can be a powerful tool for this

6.0001 LECTURE 6 50

```
def isPal(x):
 temp = x
 temp.reverse
 if temp == x:
 return True
 else:
 return False

print(isPal(list('abcba'))) returnsTrue
print(isPal(list('palinnilap'))) returnsTrue
print(isPal('ab')) XXX
```

6.0001 LECTURE 6 51

## STEPPING THROUGH THE TESTS

- suppose we run this code:
  - we try the input 'abcba', which succeeds
  - we try the input 'palinnilap', which succeeds
  - but we try the input 'ab', which also 'succeeds'
- let's use bisection search to isolate bug(s)
- pick a spot about halfway through code, and devise experiment
  - pick a spot where easy to examine intermediate values

6.0001 LECTURE 6

52

```
def isPal(x):
 temp = x
 temp.reverse
 print(temp, x)
 if temp == x:
 return True
 else:
 return False
```

```
print(isPal(list('abcba'))) prints ['a', 'b', 'c', 'b', 'a'] ['a', 'b', 'c', 'b', 'a']
print(isPal('ab')) prints XXX ['a', 'b']
```

53

## STEPPING THROUGH



- at this point in the code, we expect (for our example of 'ab') that x should be ['a', 'b'], and temp should be ['b', 'a'], however they both have the value ['a', 'b']
- so let's add another print statement, earlier in the code

6.0001 LECTURE 6

54

```
def isPal(x):
 temp = x
 print('before reverse', temp, x)
 temp.reverse
 print('after reverse', temp, x)
 if temp == x:
 return True
 else:
 return False
```

```
print(isPal(list('abcba'))) before ['a', 'b', 'c', 'b', 'a'] ['a', 'b', 'c', 'b', 'a']
 after ['a', 'b', 'c', 'b', 'a'] ['a', 'b', 'c', 'b', 'a']
print(isPal('ab')) before ['a', 'b'] ['a', 'b']
 after XXX ['a', 'b']
```

55

## STEPPING THROUGH



- we see that temp has the **same value** before and after the call to reverse
- if we look at our code, we realize we have committed a standard bug – we forgot to actually invoke the reverse method
  - need `temp.reverse()`
- so let's make that change and try again

6.0001 LECTURE 6

56

```
def isPal(x):
 temp = x
 print('before reverse', temp, x)
 temp.reverse()
 print('after reverse', temp, x)
 if temp == x:
 return True
 else:
 return False
```

```
print(isPal(list('abcba'))) before ['a', 'b', 'c', 'b', 'a'] ['a', 'b', 'c', 'b', 'a']
 after ['a', 'b', 'c', 'b', 'a'] ['a', 'b', 'c', 'b', 'a']
print(isPal('ab')) before ['a', 'b'] ['a', 'b']
 after XXX ['a', 'b']
```

57

## STEPPING THROUGH



- but now when we run on our simple example, both `x` and `temp` have been reversed!!
- we have also narrowed down this bug to a single line. The error must be in the reverse step
- in fact, we have an aliasing bug – reversing `temp` has also caused `x` to be reversed
  - because they are referring to the same object

6.0001 LECTURE 6

58

```
def isPal(x):
 temp = x[:] ←
 print('before reverse', temp, x) ←
 temp.reverse() ←
 print('after reverse', temp, x) ←
 if temp == x:
 return True
 else:
 return False

print(isPal(list('abcba'))) before['a', 'b', 'c', 'b', 'a'] after['a', 'b', 'c', 'b', 'a']
print(isPal('ab')) before['a', 'b'] after['b', 'a']
```

6.0001 LECTURE 6

59

## STEPPING THROUGH



- now running this shows that before the reverse step, the two variables have the same form, but afterwards only `temp` is reversed.
- we can now go back and check that our other test cases still work correctly

6.0001 LECTURE 6

60

## SOME PRAGMATIC HINTS



- look for the usual suspects
- ask why the code is doing what it is, not why it is not doing what you want
- the bug is probably not where you think it is – eliminate locations
- explain the problem to someone else
- don't believe the documentation
- take a break and come back to the bug later

6.0001 LECTURE 6

61

## EXCEPTIONS, ASSERTIONS

6.0001 LECTURE 6

62

## UNEXPECTED CONDITIONS

- what happens when procedure execution hits an **unexpected condition**?
- get an **exception**... to what was expected
  - trying to access beyond list limits  
`test = [1, 7, 4]` → `IndexError`
  - trying to convert an inappropriate type  
`int(test)` → `TypeError`
  - referencing a non-existing variable  
`a` → `NameError`
  - mixing data types without coercion  
`'a' / 4` → `TypeError`

6.0001 LECTURE 6

63

## OTHER EXCEPTIONS

already seen common error types:

- SyntaxError: Python can't parse program
- NameError: local or global name not found
- AttributeError: attribute reference fails
- TypeError: operand doesn't have correct type
- ValueError: operand type okay, but value is illegal
- IOError: IO system reports malfunction (e.g. file not found)

## HANDLING SPECIFIC EXCEPTIONS

have **separate except clauses** to deal with particular types of exceptions

```

try:
 a = int(input("Tell me one number: "))
 b = int(input("Tell me another number: "))
 print("a/b = ", a/b)
 print("a+b = ", a+b)
except ValueError:
 print("Could not convert to a number.") only execute if these errors come up
except ZeroDivisionError:
 print("Can't divide by zero")
except:
 print("Something went very wrong.") for all other errors

```

## WHAT TO DO WITH EXCEPTIONS?

- fail silently** – substitute default values or just continue
  - bad idea!** user gets no warning
- return an **"error" value**
  - complicates code having to check for a special value
- stop execution, **signal error** condition

```

raise <exceptionName>(<arguments>)
raise ValueError("something is wrong")

```

*Keyword: name of error you want to raise*

*optional, usually a string message*

## HANDLING EXCEPTIONS

Python code can provide **handlers** for exceptions

```

try:
 a = int(input("Tell me one number: "))
 b = int(input("Tell me another number: "))
 print(a/b)
except:
 print("Bug in user input.")

```

exceptions **raised** by any statement in body of **try** are **handled** by the **except** statement and execution continues with the body of the except statement

## OTHER EXCEPTIONS

- else:**
  - body of this is executed when execution of associated try body **completes with no exceptions**
- finally:**
  - body of this is **always executed** after try, else and except clauses, even if they raised another error or executed a break, continue or return
  - useful for clean-up code that should be run no matter what else happened (e.g. close a file)

## EXAMPLE: RAISING AN EXCEPTION

```

def get_ratios(L1, L2):
 ratios = []
 for index in range(len(L1)):
 try:
 ratios.append(L1[index]/L2[index])
 except ZeroDivisionError:
 ratios.append(float('nan')) #nan = not a number
 except:
 raise ValueError('get ratios called with bad arg')
 else:
 print("success")
 finally:
 print("executed no matter what!")
 return ratios

```

*manage flow of program by raising own error*

## EXAMPLE OF EXCEPTIONS

- assume we are **given a class list** for a subject: each entry is a list of two parts
    - a list of first and last name for a student
    - a list of grades on assignments
- ```
test_grades = [[[ 'peter', 'parker' ], [ 10.0, 5.0, 85.0 ] ],
               [ [ 'bruce', 'wayne' ], [ 10.0, 8.0, 74.0 ] ]]
```
- create a **new class list**, with name, grades, and an average


```
[[[ 'peter', 'parker' ], [ 10.0, 5.0, 85.0 ], 33.33333 ],
       [ [ 'bruce', 'wayne' ], [ 10.0, 8.0, 74.0 ], 30.666667 ]]
```

6.0001 LECTURE 6

72

EXAMPLE

CODE

```
[[[ 'peter', 'parker' ], [ 10.0, 5.0, 85.0 ]],
 [[ 'bruce', 'wayne' ], [ 10.0, 8.0, 74.0 ]]]
```

```
def get_stats(class_list):
    new_stats = []
    for elt in class_list:
        new_stats.append([elt[0], elt[1], avg(elt[1])])
    return new_stats

def avg(grades):
    return sum(grades)/len(grades)
```

6.0001 LECTURE 6

73

ERROR IF NO GRADE FOR A STUDENT

- if one or more students **don't have any grades**, get an error


```
test_grades = [[[ 'peter', 'parker' ], [ 10.0, 5.0, 85.0 ]],
                     [ [ 'bruce', 'wayne' ], [ 10.0, 8.0, 74.0 ] ],
                     [ [ 'captain', 'america' ], [ 8.0, 10.0, 96.0 ] ],
                     [ [ 'thor' ], [] ]]
```
- get zeroDivisionError: float division by zero because try to**
`return sum(grades)/len(grades)`
length is 0

6.0001 LECTURE 6

74

OPTION 1: FLAG THE ERROR BY PRINTING A MESSAGE

- decide to **notify** that something went wrong with a msg


```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
```
 - running on some test data gives


```
warning: no grades data
```

flagged the error
- ```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 33.3333333],
 [['bruce', 'wayne'], [10.0, 8.0, 74.0], 30.66666666],
 [['captain', 'america'], [8.0, 10.0, 96.0], 38.0],
 [['thor'], []]]
```
- because avg did  
 not return anything  
 in the except

6.0001 LECTURE 6

75

## OPTION 2: CHANGE THE POLICY

- decide that a student with no grades gets a **zero**

```
def avg(grades):
 try:
 return sum(grades)/len(grades)
 except ZeroDivisionError:
 print('warning: no grades data')
 return 0.0
```
  - running on some test data gives
 

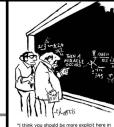
```
warning: no grades data
```

still flag the error
- ```
[[[ 'peter', 'parker' ], [ 10.0, 5.0, 85.0 ], 33.333333 ],
 [[ 'bruce', 'wayne' ], [ 10.0, 8.0, 74.0 ], 30.666666 ],
 [[ 'captain', 'america' ], [ 8.0, 10.0, 96.0 ], 38.0 ],
 [[ 'thor' ], [] ]]
```
- now avg returns 0

6.0001 LECTURE 6

76

ASSERTIONS

- 
- want to be sure that **assumptions** on state of computation are what we expected
 - use an **assert statement** to raise an `AssertionError` exception if assumptions not met
 - an example of good **defensive programming**

6.0001 LECTURE 6

77

EXAMPLE

```
def avg(grades):
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

function ends
immediately if
assertion not met

- raises an `AssertionError` if it is given an empty list for `grades`
- otherwise runs ok

6.0001 LECTURE 6

78

ASSERTIONS AS DEFENSIVE PROGRAMMING

- assertions don't allow a programmer to control response to unexpected conditions
- ensure that `execution halts` whenever an expected condition is not met
- typically used to `check inputs` to functions, but can be used anywhere
- can be used to `check outputs` of a function to avoid propagating bad values
- can make it easier to locate a source of a bug

6.0001 LECTURE 6

79

WHERE TO USE ASSERTIONS?

- goal is to spot bugs as soon as introduced and make clear where they happened
- use as a `supplement` to testing
- raise `exceptions` if user supplies `bad data input`
- use `assertions` to
 - check `types` of arguments or values
 - check that `invariants` on data structures are met
 - check `constraints` on return values
 - check for `violations` of constraints on procedure (e.g. no duplicates in a list)

6.0001 LECTURE 6

80