

# DICTIONARIES, DEBUGGING, EXCEPTIONS

(download slides and .py files to follow along!)

---

6.0001 LECTURE 6

# LAST TIME

---

- tuples - immutable
- lists - mutable
- aliasing, cloning
- mutability side effects

# TODAY

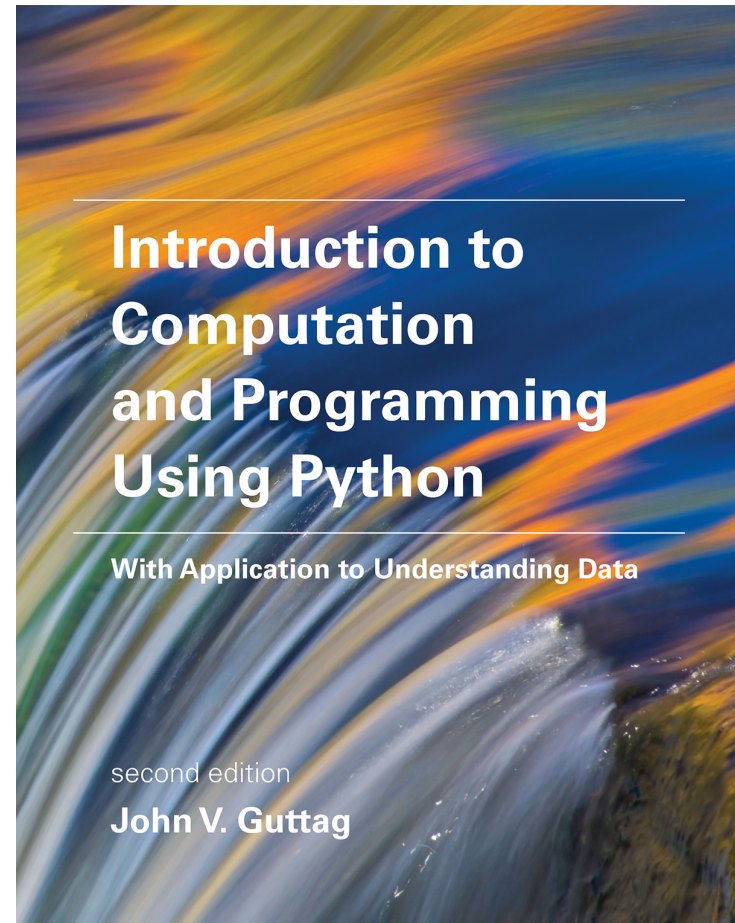
---

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

# Assigned Reading

---

- Section 5.6
- Chapter 6
- Chapter 7



# DICTIONARIES

---

# HOW TO STORE STUDENT INFO

---

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Matt', 'Katy']
```

```
grade = ['B', 'A+', 'A', 'A']
```

```
course = [2.00, 6.00, 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 info for a different person

# HOW TO UPDATE/RETRIEVE 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 to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

# A BETTER AND CLEANER WAY – A DICTIONARY

---

- nice 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	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
...	...

custom  
index by  
label  
element



# A PYTHON DICTIONARY

- store pairs of data
  - key
  - value (any object type)

'Ana'	'B'
'Matt'	'A'
'John'	'A+'
'Katy'	'A'

custom  
index by  
label

element

my\_dict = { } *empty dictionary*

grades = { 'Ana': 'B', 'John': 'A+', 'Matt': 'A', 'Katy': 'A' }

↑      ↑      ↑      ↑      ↑      ↑      ↑      ↑  
key1 val1 key2 val2 key3 val3 key4 val4

# 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'
'Matt'	'A'
'John'	'A+'
'Katy'	'A'

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

```
grades['John']      → evaluates to 'A+'
```

```
grades['Laura']     → gives a KeyError
```

# DICTIONARY OPERATIONS

---

'Ana'	'B'
'Matt'	'A'
'John'	'A+'
'Katy'	'A'
'Laura'	'A'

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

- **add** an entry

```
grades['Laura'] = 'A'
```

- **test** if key in dictionary

```
'John' in grades
```

→ returns True

```
'Daniel' in grades
```

→ returns False

- **delete** entry

```
del(grades['Ana'])
```

# DICTIONARY OPERATIONS

---

'Ana'	'B'
'Matt'	'A'
'John'	'A+'
'Katy'	'A'

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

- get an **iterable that acts like a tuple of all keys**

```
grades.keys()
```

→ returns `dict_keys(['Matt', 'Katy', 'John', 'Ana'])`

*no guaranteed  
order*

- get an **iterable that acts like a tuple of all values**

```
grades.values()
```

→ returns `dict_values(['A', 'A+', 'B', 'A'])`

*no guaranteed  
order*

# DICTIONARY KEYS and 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 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]}
```

# list

## vs

# dict

---

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

- **matches** “keys” to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

# EXAMPLE: THREE FUNCTIONS TO ANALYZE SONG LYRICS

---

- 1) create a **frequency dictionary** mapping `str:int`
- 2) find **word that occurs the most** and how many times
  - use a list, in case there is more than one word
  - return a tuple `(list,int)` for `(words_list, highest_freq)`
- 3) 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.

# CREATING A DICTIONARY

---

```
def generate_word_dict(song):  
    words_list = song.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
```

*iterate over list of  
words in song*

*if word in dict, increase #  
times you've seen it*

*if word not in dict, seen  
word once*



# 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*

*all words that  
have that freq*

# LEVERAGING DICTIONARY 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
```

*common use of Boolean is a  
flag to stay/exit from a loop*

*finding freqs  
higher than atleast*

*mutate dict*

# FIBONACCI RECURSIVE CODE (MULTIPLE BASE CASES)

---

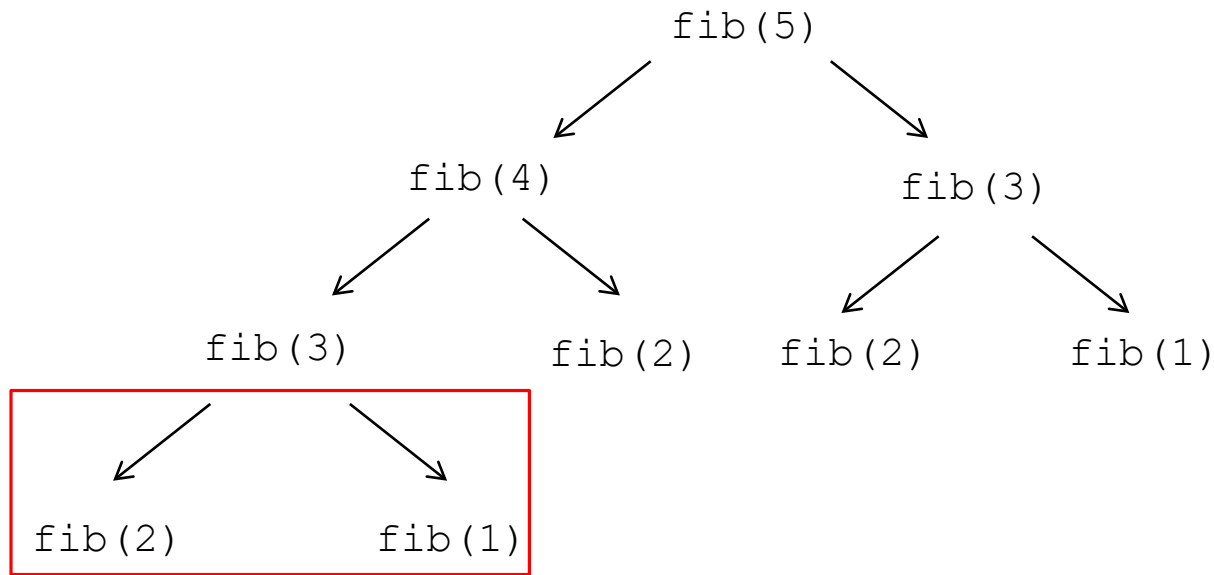
```
def fib(n):  
    if n == 1:  
        return 1  
    elif n == 2:  
        return 2  
    else:  
        return fib(n-1) + fib(n-2)
```

- Two base cases
- Calls itself twice
- This code is inefficient

# INEFFICIENT FIBONACCI

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

---



base cases

- **recalculating** the same values many times!
- could keep **track** of already calculated values

# FIBONACCI WITH A DICTIONARY (aka MEMOIZATION)

---

```
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  
d = {1:1, 2:2}  
print(fib_efficient(6, d))
```

*Initialize dict  
with base cases*

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

# HOW MUCH MORE EFFICIENT?

---

- calling `fib(34)` results in **11,405,773** recursive calls
- calling `fib_efficient(34)` results in **65** recursive calls!
- very efficient!
- this only works for **functions without side effects** (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)



# TESTING, DEBUGGING

---



## DEFENSIVE PROGRAMMING

- Write **specifications** for functions
  - **Modularize** programs
  - Check **conditions** on inputs/outputs (assertions)
- 
- ```
graph TD; A[DEFENSIVE PROGRAMMING] --> B[TESTING/VALIDATION]; A --> C[DEBUGGING];
```

### TESTING/VALIDATION

- **Compare** input/output pairs to specification
- “It’s not 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?”



# SET YOURSELF UP FOR EASY TESTING AND DEBUGGING

---

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

# 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

# CLASSES OF TESTS

---



## ■ Unit testing

- validate each piece of program
- **testing each function** separately

## ■ Regression testing

- add test for bugs as you find them
- **catch reintroduced** errors that were previously fixed

## ■ Integration testing

- does **overall program** work?
- tend to rush to do this

# 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

# 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)

# 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 """
```

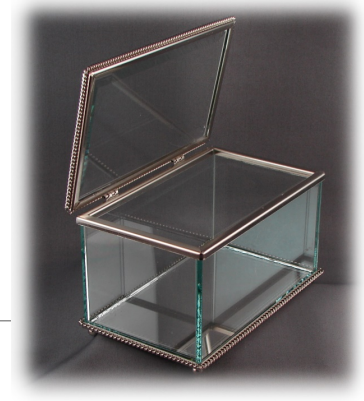
| CASE                   | x             | eps           |
|------------------------|---------------|---------------|
| boundary               | 0             | 0.0001        |
| perfect square         | 25            | 0.0001        |
| less than 1            | 0.05          | 0.0001        |
| irrational 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     |

# 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

# 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 `abs(-1)` incorrectly returns -1
- should still test boundary cases



# BUGS



# DEBUGGING

---

- 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 casessteep 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

# 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`

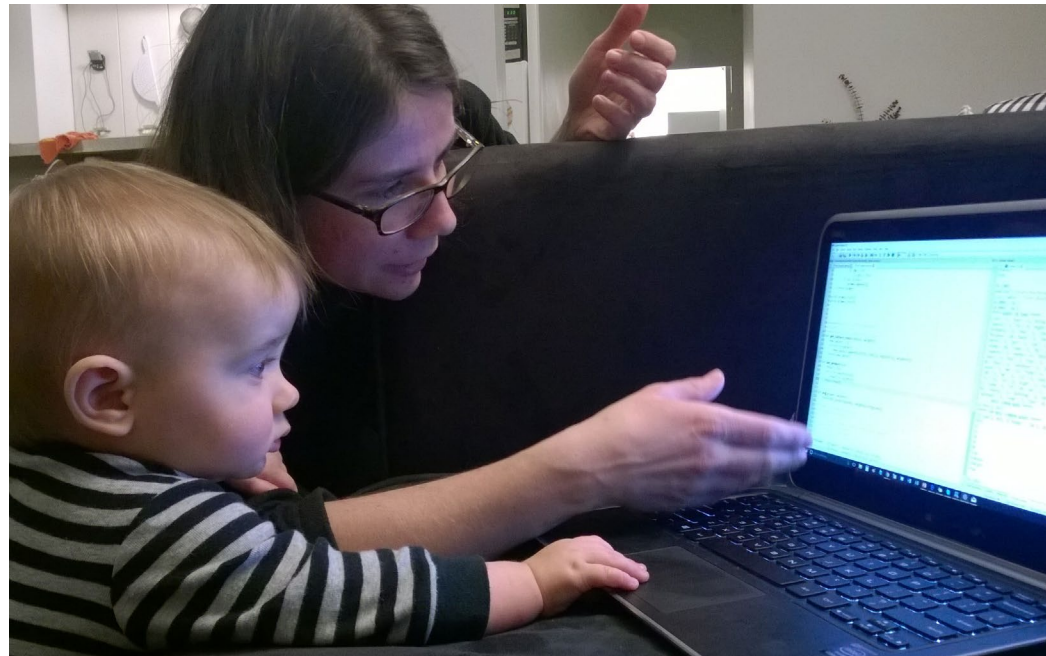
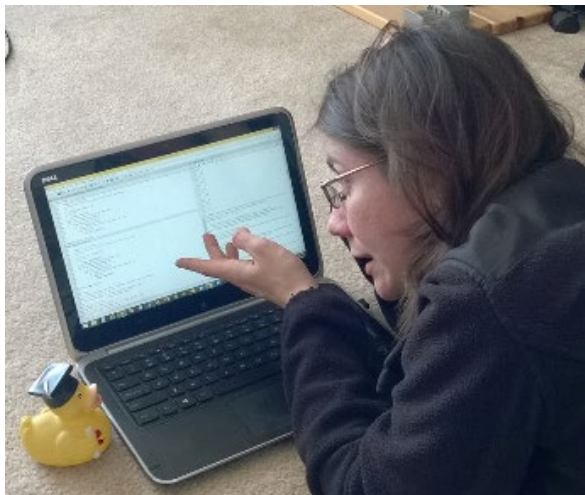
- forgetting to close parenthesis, quotation, etc.

`a = len([1,2,3]`  
`print(a)`                                              → `SyntaxError`

# LOGIC ERRORS - HARD

---

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



# DEBUGGING STEPS

---

- **study** program code
  - don't ask what is wrong
  - ask how did I get the unexpected result
  - is it part of a family?
- **scientific method**
  - study available data
  - form hypothesis
  - repeatable experiments
  - pick simplest input to test with

# PRINT STATEMENTS

---

- good way to **test hypothesis**
- when to print
  - enter function
  - parameters
  - function results
- use **bisection method**
  - put print halfway in code
  - decide where bug may be depending on values

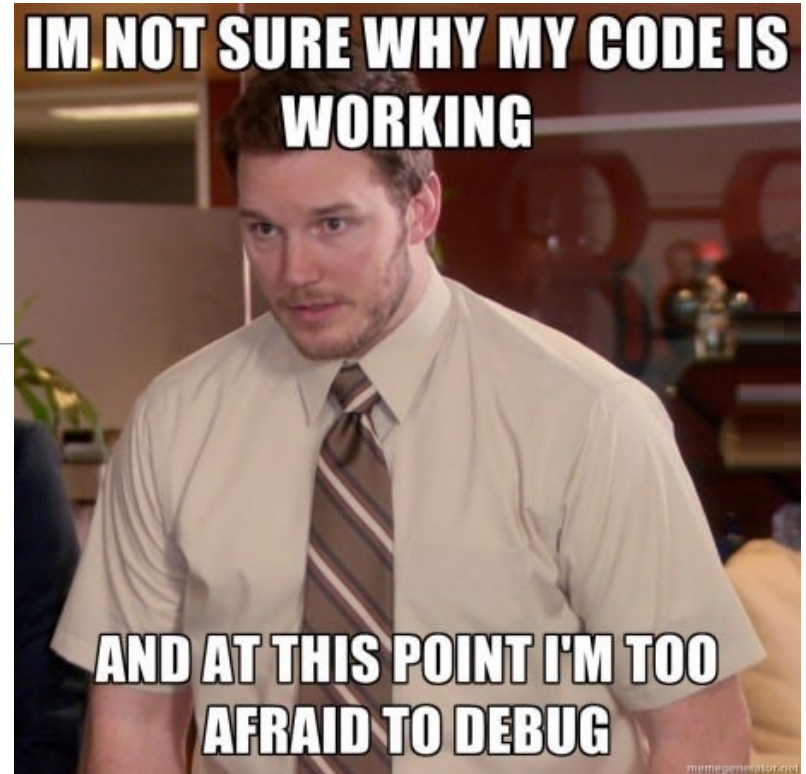
# 5 Minute Break



Actual  
programming



Debating for  
30 minutes on  
how to name a  
variable



When my code somehow just works

# EXCEPTIONS, ASSERTIONS

---



# EXCEPTIONS AND ASSERTIONS

---

- 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]
```

```
test[4]
```

→ IndexError

- trying to convert an inappropriate type

```
int(test)
```

→ TypeError

- referencing a non-existing variable

```
a
```

→ NameError

- mixing data types without coercion

```
'5' / 4
```

→ TypeError

# OTHER TYPES OF 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)

# DEALING WITH 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

# HANDLING SPECIFIC EXCEPTIONS

---

- have **separate except clauses** to deal with a particular type of exception

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.")
```

```
except ZeroDivisionError:
    print("Can't divide by zero")
```

```
except:
    print("Something went very wrong.")
```

*only execute  
if these errors  
come up*

*for all  
other  
errors*

# 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)

# WHAT TO DO WITH EXCEPTIONS?

---

- what to do when encounter an error?
- **fail silently:**
  - substitute default values or just continue
  - bad idea! user gets no warning
- return an **“error” value**
  - what value to choose?
  - complicates code having to check for a special value
- stop execution, **signal error** condition
  - in Python: **raise an exception**  
`raise ValueError("something is wrong")`

# EXAMPLE: RAISING AN EXCEPTION

---

```
def get_ratios(L1, L2):  
    """ Assumes: L1 and L2 are lists of equal length of numbers  
        Returns: a list containing L1[i]/L2[i] """  
    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')  
    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'], [80.0, 70.0, 85.0]],  
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],  
 [['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]]
```



# EXAMPLE

## CODE

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],  
 [['bruce', 'wayne'], [100.0, 80.0, 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)
```

# 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*

# 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], 15.41666666],  
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],  
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],  
[['thor'], [], None]]
```

*because avg did  
not return anything  
in the except*

# 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
```

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],  
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],  
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],  
[['thor'], [], 0.0]]
```

*still flag the error*

*now avg returns 0*

# ASSERTIONS

---

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

# 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

# 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

# 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 users 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)