Sample solutions

0-basics (Score: 10.0 / 10.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 1.0 / 1.0)
4. Test cell (Score: 2.0 / 2.0)
5. Test cell (Score: 1.0 / 1.0)
6. Test cell (Score: 1.0 / 1.0)
7. Test cell (Score: 1.0 / 1.0)
8. Test cell (Score: 2.0 / 2.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Python review: Values, variables, types, lists, and strings

These first few notebooks are a set of exercises with two goals:

1. Review the basics of Python
2. Familiarize you with Jupyter

Regarding the first goal, these initial notebooks cover material we think you should already know from Chris Simpkins's (https://www.cc.gatech.edu/~simpkins/) Python Bootcamp (https://www.cc.gatech.edu/~simpkins/teaching/python-bootcamp/syllabus.html). It is based specifically on his offering to incoming students of the Georgia Tech MS Analytics in Fall 2016 (https://www.cc.gatech.edu/~simpkins/teaching/python-bootcamp/august2016.html).

Regarding the second goal, you'll observe that the bootcamp has each student install and work directly with the Python interpreter, which runs locally on his or her machine (e.g., see Slide 5 of Chris's intro (https://www.cc.gatech.edu/~simpkins/teaching/python-bootcamp/slides/intro-python.html)). But in this course, we are using Jupyter Notebooks as the development environment. You can think of a Jupyter notebook as a web-based "skin" for running a Python interpreter---possibly hosted on a remote server, which is the case in this course. Here is a good tutorial on Jupyter (https://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook).

> **Note for OMSA (https://pe.gatech.edu/master-science-degrees/online-master-science-analytics) students.** In this course we assume you are using Vocareum's deployment (https://www.vocareum.com/) of Jupyter. You also have an option to use other Jupyter environments, including installing and running Jupyter on your own system. We can't provide technical support to you if you choose to go those routes, but if you'd like to do that anyway, we recommend Microsoft Azure Notebooks (https://notebooks.azure.com/) as a web-hosted option, which we use in the on-campus class, or the Continuum Analytics Anaconda distribution (https://www.continuum.io/downloads) as a locally installed option.

Study hint: Read the test code! You'll notice that most of the exercises below have a place for you to code up your answer followed by a "test cell." That's a code cell that checks the output of your code to see whether it appears to produce correct results. You can often learn a lot by reading the test code. In fact, sometimes it gives you a hint about how to approach the problem. As such, we encourage you to try to read the test cells even if they seem cryptic, which is deliberate!

Exercise 0 (1 point). Run the code cell below. It should display the output string, `Hello, world!`.

In [1]:

Grade cell: `hello_world_test` Score: 1.0 / 1.0 (Top)

```
print("Hello, world!")
```

```
Hello, world!
```

Exercise 1 (`x_float_test`: 1 point). Create a variable named `x_float` whose numerical value is one (1) and whose type is *floating-point*.

In [2]:

Student's answer (Top)

```
x_float = 1.0
```

In [3]:

Grade cell: `x_float_test` Score: 1.0 / 1.0 (Top)

```
# `x_float_test`: Test cell
assert x_float == 1
assert type(x_float) is float
print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 2 (`strcat_ba_test`: 1 point). Complete the following function, `strcat_ba(a, b)`, so that given two strings, `a` and `b`, it returns the concatenation of `b` followed by `a` (pay attention to the order in these instructions!).

In [4]:

Student's answer (Top)

```
def strcat_ba(a, b):
    assert type(a) is str
    assert type(b) is str
    return b + a
```

In [5]:

Grade cell: `strcat_ba_test` Score: 1.0 / 1.0 (Top)

```
# `strcat_ba_test`: Test cell

# Workaround:  # Python 3.5.2 does not have `random.choices()` (available in 3.6+)
def random_letter():
    from random import choice
    return choice('abcdefghijklmnopqrstuvwxyz')

def random_string(n, fun=random_letter):
    return ''.join([str(fun()) for _ in range(n)])

a = random_string(5)
b = random_string(3)
c = strcat_ba(a, b)
print('strcat_ba("{}", "{}") == "{}"'.format(a, b, c))
assert len(c) == len(a) + len(b)
assert c[:len(b)] == b
assert c[-len(a):] == a
print("\n(Passed!)")
```

```
strcat_ba("qmlga", "yzj") == "yzjqmlga"

(Passed!)
```

Exercise 3 (`strcat_list_test`: 2 points). Complete the following function, `strcat_list(L)`, which generalizes the previous function: given a *list* of strings, `L[:]`, returns the concatenation of the strings in reverse order. For example:

```
strcat_list(['abc', 'def', 'ghi']) == 'ghidefabc'
```

In [6]:

Student's answer (Top)

```
def strcat_list(L):
    assert type(L) is list
    return ''.join(L[::-1])
```

In [7]:

Grade cell: `strcat_list_test` Score: 2.0 / 2.0 (Top)

```
# `strcat_list_test`: Test cell
n = 3
nL = 6
L = [random_string(n) for _ in range(nL)]
Lc = strcat_list(L)
```

```
print('L == {}'.format(L))
print('strcat_list(L) == \'{}\''.format(Lc))
assert all([Lc[i*n:(i+1)*n] == L[nL-i-1] for i, x in zip(range(nL), L)])
print("\n(Passed!)")
```

```
L == ['vsp', 'yyn', 'yoh', 'iqv', 'kii', 'nby']
strcat_list(L) == 'nbykiiiqvyohyynvsp'

(Passed!)
```

Exercise 4 (`floor_fraction_test`: 1 point). Suppose you are given two variables, `a` and `b`, whose values are the real numbers, $a \geq 0$ (non-negative) and $b > 0$ (positive). Complete the function, `floor_fraction(a, b)` so that it returns $\left\lfloor \frac{a}{b} \right\rfloor$, that is, the *floor* of $\frac{a}{b}$. The *type* of the returned value must be `int` (an integer).

In [8]:

Student's answer (Top)

```
def is_number(x):
    """Returns `True` if `x` is a number-like type, e.g., `int`, `float`, `Decimal()`,
 ..."""
    from numbers import Number
    return isinstance(x, Number)

def floor_fraction(a, b):
    assert is_number(a) and a >= 0
    assert is_number(b) and b > 0
    return int(a / b) # Alternative: Use the Python library call, `int(math.floor(a/b))
`
```

In [9]:

Grade cell: `floor_fraction_test` Score: 1.0 / 1.0 (Top)

```
# `floor_fraction_test`: Test cell
from random import random
a = random()
b = random()
c = floor_fraction(a, b)

print('floor_fraction({}, {}) == floor({}) == {}'.format(a, b, a/b, c))
assert b*c <= a <= b*(c+1)
assert type(c) is int
print('\n(Passed!)')
```

```
floor_fraction(0.1322891757479948, 0.8374075207655717) == floor(0.15797466880528357) == 0

(Passed!)
```

Exercise 5 (`ceiling_fraction_test`: 1 point). Complete the function, `ceiling_fraction(a, b)`, which for any numeric inputs, `a` and `b`, corresponding to real numbers, $a \geq 0$ and $b > 0$, returns $\left\lceil \frac{a}{b} \right\rceil$, that is, the *ceiling* of $\frac{a}{b}$. The type of the returned value must be `int`.

In [10]:

Student's answer (Top)

```
def ceiling_fraction(a, b):
    assert is_number(a) and a >= 0
    assert is_number(b) and b > 0
    r = a/b - floor_fraction(a, b)
    return int(a/b + int(r > 0))
    # Alternative: Use Python's library call, `int(math.ceiling(a/b))`
```

In [11]:

Grade cell: `ceiling_fraction_test` Score: 1.0 / 1.0 (Top)

```
# `ceiling_fraction_test`: Test cell
from random import random
a = random()
b = random()
c = ceiling_fraction(a, b)
print('ceiling_fraction({}, {}) == ceiling({}) == {}'.format(a, b, a/b, c))
assert b*(c-1) <= a <= b*c
assert type(c) is int
print("\n(Passed!)")
```

```
ceiling_fraction(0.2693079078811397, 0.9917083503230812) == ceiling(0.2715595848249174) =
= 1

(Passed!)
```

Exercise 6 (`report_exam_avg_test`: 1 point). Let `a`, `b`, and `c` represent three exam scores as numerical values. Complete the function, `report_exam_avg(a, b, c)` so that it computes the average score (equally weighted) and returns the string, `'Your average score is: XX'`, where `XX` is the average rounded to one decimal place. For example:

```
report_exam_avg(100, 95, 80) == 'Your average score: 91.7'
```

In [12]:

Student's answer (Top)

```
def report_exam_avg(a, b, c):
    assert is_number(a) and is_number(b) and is_number(c)
    m = (a + b + c) / 3
    return 'Your average score: {:.1f}'.format(m)
```

In [13]:

Grade cell: `report_exam_avg_test` Score: 1.0 / 1.0 (Top)

```
# `report_exam_avg_test`: Test cell
msg = report_exam_avg(100, 95, 80)
print(msg)
assert msg == 'Your average score: 91.7'

print("Checking some additional randomly generated cases:")
for _ in range(10):
    ex1 = random() * 100
    ex2 = random() * 100
    ex3 = random() * 100
    msg = report_exam_avg(ex1, ex2, ex3)
    ex_rounded_avg = float(msg.split()[-1])
    abs_err = abs(ex_rounded_avg*3 - (ex1 + ex2 + ex3)) / 3
    print("{}, {}, {} -> '{}' [{}]".format(ex1, ex2, ex3, msg, abs_err))
    assert abs_err <= 0.05

print("\n(Passed!)")
```

```
Your average score: 91.7
Checking some additional randomly generated cases:
44.17850353375138, 12.88968726663694, 57.78837019116971 -> 'Your average score: 38.3' [0.
014479669480650159]
50.318162124565944, 96.6537804184974, 74.47161821084232 -> 'Your average score: 73.8' [0.
014520251301898194]
16.842460366963763, 68.89366751649963, 32.69410415655918 -> 'Your average score: 39.5'
[0.023255986659142042]
7.927805923019459, 49.358814693750915, 52.557899732982506 -> 'Your average score: 36.6'
[0.014840116584290778]
46.504302721345184, 62.75854112568808, 10.899164353783153 -> 'Your average score: 40.1'
[0.045997266394531756]
85.77415479077843, 37.09171308319815, 30.14936020349195 -> 'Your average score: 51.0' [0.
0050760258228403 17]
85.43086416065485, 52.22980329251694, 77.52770911281944 -> 'Your average score: 71.7' [0.
029458855330403 82]
70.76253130781427, 1.6998159529025725, 59.439635910063394 -> 'Your average score: 44.0'
[0.03267227640659106]
40.49103214035552, 84.58373490752865, 29.443292854116088 -> 'Your average score: 51.5'
[0.006019967333420 1025]
88.29426488701392, 33.13383413804565, 26.800373444672932 -> 'Your average score: 49.4'
[0.009490823244173422]

(Passed!)
```

Exercise 7 (`count_word_lengths_test`: 2 points). Write a function `count_word_lengths(s)` that, given a string consisting of words separated by spaces, returns a list containing the length of each word. Words will consist of lowercase alphabetic characters, and they may be separated by multiple consecutive spaces. If a string is empty or has no spaces, the function should return an empty list.

For instance, in this code sample,

```
count_word_lengths('the quick  brown   fox jumped over     the lazy  dog') == [3, 5, 5, 3, 6, 4, 3
, 4, 3]`
```

the input string consists of nine (9) words whose respective lengths are shown in the list.

In [14]:

Student's answer (Top)

```
def count_word_lengths(s):
    assert all([x.isalpha() or x == ' ' for x in s])
    assert type(s) is str
    return [len(x) for x in s.split()]
```

In [15]:

Grade cell: `count_word_lengths_test` Score: 2.0 / 2.0 (Top)

```
# `count_word_lengths_test`: Test cell

# Test 1: Example
qbf_str = 'the quick brown fox jumped over the lazy dog'
qbf_lens = count_word_lengths(qbf_str)
print("Test 1: count_word_lengths('{}') == {}".format(qbf_str, qbf_lens))
assert qbf_lens == [3, 5, 5, 3, 6, 4, 3, 4, 3]

# Test 2: Random strings
from random import choice # 3.5.2 does not have `choices()` (available in 3.6+)
#return ''.join([choice('abcdefghijklmnopqrstuvwxyz') for _ in range(n)])

def random_letter_or_space(pr_space=0.15):
    from random import choice, random
    is_space = (random() <= pr_space)
    if is_space:
        return ' '
    return random_letter()

S_LEN = 40
W_SPACE = 1 / 6
rand_str = random_string(S_LEN, fun=random_letter_or_space)
rand_lens = count_word_lengths(rand_str)
print("Test 2: count_word_lengths('{}') == '{}'".format(rand_str, rand_lens))
c = 0
while c < len(rand_str) and rand_str[c] == ' ':
    c += 1
for k in rand_lens:
    print("  => '{}'".format (rand_str[c:c+k]))
    assert (c+k) == len(rand_str) or rand_str[c+k] == ' '
    c += k
    while c < len(rand_str) and rand_str[c] == ' ':
        c += 1

# Test 3: Empty string
print("Test 3: Empty strings...")
assert count_word_lengths('') == []
assert count_word_lengths('   ') == []

print("\n(Passed!)")
```

```
Test 1: count_word_lengths('the quick brown fox jumped over the lazy dog') == [3, 5, 5,
3, 6, 4, 3, 4, 3]
Test 2: count_word_lengths(' k piiwa  ewumu zuhhtf faq udz rrhl ed q') == '[1, 5, 5, 6,
3, 3, 4, 2, 1]'
  => 'k'
  => 'piiwa'
  => 'ewumu'
  => 'zuhhtf'
  => 'faq'
  => 'udz'
  => 'rrhl'
  => 'ed'
  => 'q'
Test 3: Empty strings...

(Passed!)
```

In [16]:

1-collections (Score: 8.0 / 8.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 2.0 / 2.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 2.0 / 2.0)
5. Test cell (Score: 1.0 / 1.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Python review: Basic collections of values

This notebook continues the review of Python basics based on Chris Simpkins's (https://www.cc.gatech.edu/~simpkins/) Python Bootcamp (https://www.cc.gatech.edu/~simpkins/teaching/python-bootcamp/syllabus.html). The focus here is on basic collections: tuples, dictionaries, and sets.

Exercise 0 (`minmax_test`: 1 point). Complete the function `minmax(L)`, which takes a list `L` and returns a pair---that is, 2-element Python tuple, or "2-tuple"---whose first element is the minimum value in the list and whose second element is the maximum. For instance:

```
minmax([8, 7, 2, 5, 1]) == (1, 8)
```

In [1]:

Student's answer (Top)

```
def minmax(L):
    assert hasattr(L, "__iter__")
    return (min(L), max(L))
```

In [2]:

Grade cell: `minmax_test` Score: 1.0 / 1.0 (Top)

```
# `minmax_test`: Test cell

L = [8, 7, 2, 5, 1]
mmL = minmax(L)
mmL_true = (1, 8)
print("minmax({}) -> {} [True: {}]".format(L, mmL, mmL_true))
assert type(mmL) is tuple and mmL == (1, 8)

from random import sample
L = sample(range(1000), 10)
mmL = minmax(L)
L_s = sorted(L)
mmL_true = (L_s[0], L_s[-1])
print("minmax({}) -> {} [True: {}]".format(L, mmL, mmL_true))
assert mmL == mmL_true

print("\n(Passed!)")
```

```
minmax([8, 7, 2, 5, 1]) -> (1, 8) [True: (1, 8)]
minmax([900, 478, 446, 710, 463, 858, 124, 438, 633, 285]) -> (124, 900) [True: (124, 90
0)]

(Passed!)
```

Exercise 1 (`remove_all_test`: 2 points). Complete the function `remove_all(L, x)` so that, given a list `L` and a target value `x`, it returns a *copy* of the list that excludes *all* occurrences of `x` but preserves the order of the remaining elements. For instance:

```
remove_all([1, 2, 3, 2, 4, 8, 2], 2) == [1, 3, 4, 8]
```

> **Note.** Your implementation should *not* modify the list being passed into `remove_all`.

In [3]:

Student's answer (Top)

```
def remove_all(L, x):
    assert type(L) is list and x is not None
    return [v for v in L if v != x]
```

In [4]:

Grade cell: `remove_all_test` Score: 2.0 / 2.0 (Top)

```
# `remove_all_test`: Test cell
def test_it(L, x, L_ans):
    print("Testing `remove_all({}, {})`...".format(L, x))
    print("\tTrue solution: {}".format(L_ans))
    L_copy = L.copy()
    L_rem = remove_all(L_copy, x)
    print("\tYour computed solution: {}".format(L_rem))
    assert L_copy == L, "Your code appears to modify the input list."
```

```
    assert L_rem == L_ans, "The returned list is incorrect."

# Test 1: Example
test_it([1, 2, 3, 2, 4, 8, 2], 2, [1, 3, 4, 8])

# Test 2: Random list
from random import randint
target = randint(0, 9)
L_input = []
L_ans = []
for _ in range(20):
    v = randint(0, 9)
    L_input.append(v)
    if v != target:
        L_ans.append(v)
test_it(L_input, target, L_ans)

print("\n(Passed!)")
```

```
Testing `remove_all([1, 2, 3, 2, 4, 8, 2], 2)`...
        True solution: [1, 3, 4, 8]
        Your computed solution: [1, 3, 4, 8]
Testing `remove_all([6, 6, 1, 8, 0, 4, 9, 5, 6, 1, 9, 7, 8, 2, 2, 2, 1, 5, 1, 7], 9)`...
        True solution: [6, 6, 1, 8, 0, 4, 5, 6, 1, 7, 8, 2, 2, 2, 1, 5, 1, 7]
        Your computed solution: [6, 6, 1, 8, 0, 4, 5, 6, 1, 7, 8, 2, 2, 2, 1, 5, 1, 7]

(Passed!)
```

Exercise 2 (`compress_vector_test`: 2 points). Suppose you are given a vector, `x`, containing real values that are mostly zero. For instance:

```
x = [0.0, 0.87, 0.0, 0.0, 0.0, 0.32, 0.46, 0.0, 0.0, 0.10, 0.0, 0.0]
```

Complete the function, `compress_vector(x)`, so that returns a dictionary `d` with two keys, `d['inds']` and `d['vals']`, which are lists that indicate the position and value of all the *non-zero* entries of `x`. For the previous example,

```
d['inds'] = [1, 5, 6, 9]
    d['vals'] = [0.87, 0.32, 0.46, 0.10]
```

> **Note 1.** Your implementation must *not* modify the input vector `x`.
>
> **Note 2.** If `x` contains only zero entries, `d['inds']` and `d['vals']` should be empty lists.

In [5]:

Student's answer (Top)

```
def compress_vector(x):
    assert type(x) is list
    d = {'inds': [], 'vals': []}
    for i, v in enumerate(x):
        if v != 0.0:
            d['inds'].append(i)
            d['vals'].append(v)
    return d
```

In [6]:

Grade cell: `compress_vector_test` Score: 2.0 / 2.0 (Top)

```
# `compress_vector_test`: Test cell
def check_compress_vector(x_orig):
    print("Testing `compress_vector(x={})`:".format(x_orig))
    x = x_orig.copy()
    nz = x.count(0.0)
    print("\t`x` has {} zero entries.".format(nz))
    d = compress_vector(x)
    print("\tx (after call): {}".format(x))
    print("\td: {}".format(d))
    assert x == x_orig, "Your implementation appears to modify the input."
    assert type(d) is dict, "Output type is not `dict` (a dictionary)."
    assert 'inds' in d and type(d['inds']) is list, "Output key, 'inds', does not have
 a value of type `list`."
    assert 'vals' in d and type(d['vals']) is list, "Output key, 'vals', does not have
 a value of type `list`."
    assert len(d['inds']) == len(d['vals']), "`d['inds']` and `d['vals']` are lists of
 unequal length."
    for i, v in zip(d['inds'], d['vals']):
        assert x[i] == v, "x[{}] == {} instead of {}".format(i, x[i], v)
    assert nz + len(d['vals']) == len(x), "Output may be missing values."
    assert len(d.keys()) == 2, "Output may have keys other than 'inds' and 'vals'."

# Test 1: Example
x = [0.0, 0.87, 0.0, 0.0, 0.0, 0.32, 0.46, 0.0, 0.0, 0.10, 0.0, 0.0]
check_compress_vector(x)
```

```
check_compress_vector(x)

# Test 2: Random sparse vectors
from random import random
for _ in range(3):
    print("")
    x = []
    for _ in range(20):
        if random() <= 0.8: # Make about 10% of entries zero
            v = 0.0
        else:
            v = float("{:.2f}".format(random()))
        x.append(v)
    check_compress_vector(x)

# Test 3: Empty vector
x = [0.0] * 10
check_compress_vector(x)

print("\n(Passed!)")
```

```
Testing `compress_vector(x=[0.0, 0.87, 0.0, 0.0, 0.0, 0.32, 0.46, 0.0, 0.0, 0.1, 0.0, 0.
0])`:
        `x` has 8 zero entries.
        x (after call): [0.0, 0.87, 0.0, 0.0, 0.0, 0.32, 0.46, 0.0, 0.0, 0.1, 0.0, 0.0]
        d: {'vals': [0.87, 0.32, 0.46, 0.1], 'inds': [1, 5, 6, 9]}

Testing `compress_vector(x=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.37, 0.0, 0.0,
0.0, 0.81, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])`:
        `x` has 18 zero entries.
        x (after call): [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.37, 0.0, 0.0, 0.
0, 0.81, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        d: {'vals': [0.37, 0.81], 'inds': [9, 13]}

Testing `compress_vector(x=[0.0, 0.0, 0.49, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.3, 0.0, 0.0, 0.0, 0.0, 0.71])`:
        `x` has 17 zero entries.
        x (after call): [0.0, 0.0, 0.49, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.3, 0.0, 0.0, 0.0, 0.0, 0.71]
        d: {'vals': [0.49, 0.3, 0.71], 'inds': [2, 14, 19]}

Testing `compress_vector(x=[0.0, 0.0, 0.0, 0.0, 0.0, 0.37, 0.61, 0.0, 0.0, 0.0, 0.0, 0.4
3, 0.0, 0.0, 0.32, 0.0, 0.0, 0.0, 0.0, 0.0])`:
        `x` has 16 zero entries.
        x (after call): [0.0, 0.0, 0.0, 0.0, 0.0, 0.37, 0.61, 0.0, 0.0, 0.0, 0.0, 0.43,
0.0, 0.0, 0.32, 0.0, 0.0, 0.0, 0.0, 0.0]
        d: {'vals': [0.37, 0.61, 0.43, 0.32], 'inds': [5, 6, 11, 14]}
Testing `compress_vector(x=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])`:
        `x` has 10 zero entries.
        x (after call): [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        d: {'vals': [], 'inds': []}

(Passed!)
```

Repeated indices. Consider the compressed vector data structure, `d`, in the preceding exercise, which stores a list of indices (`d['inds']`) and a list of values (`d['vals']`).

Suppose we allow duplicate indices, possibly with different values. For example:

```
d['inds'] == [0,   3,   7,   3,   3,   5, 1]
    d['vals'] == [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

In this case, the index 3 appears three times. (Also note that the indices `d['ind']` need not appear in sorted order.)

Let's adopt the convention that when there are repeated indices, the "true" value there is the *sum* of the individual values. In other words, the true vector corresponding to this example of `d` would be:

```
# ind:  0    1    2    3*    4    5    6    7
    x == [1.0, 7.0, 0.0, 11.0, 0.0, 6.0, 0.0, 3.0]
```

Exercise 3 (`decompress_vector_test`: 2 points). Complete the function `decompress_vector(d)` that takes a compressed vector `d`, which is a dictionary with keys for the indices (`inds`) and values (`vals`), and returns the corresponding full vector. For any repeated index, the values should be summed.

The function should accept an *optional* parameter, `n`, that specifies the length of the full vector. You may assume this length is at least `max(d['inds'])+1`.

In [7]:

Student's answer (Top)

```
def decompress_vector(d, n=None):
    # Checks the input
    assert type(d) is dict and 'inds' in d and 'vals' in d, "Not a dictionary or missin
```

```
g keys"
    assert type(d['inds']) is list and type(d['vals']) is list, "Not a list"
    assert len(d['inds']) == len(d['vals']), "Length mismatch"

    # Determine length of the full vector
    i_max = max(d['inds']) if d['inds'] else -1
    if n is None:
        n = i_max+1
    else:
        assert n > i_max, "Bad value for full vector length"

    x = [0.0] * n
    for i, v in zip(d['inds'], d['vals']):
        x[i] += v
    return x                                                                    \\\\
```

In [8]:

Grade cell: `decompress_vector_test` Score: 2.0 / 2.0 (Top)

```
# `decompress_vector_test`: Test cell
def check_decompress_vector(d_orig, x_true):
    print("Testing `decompress_vector(d, n)`:")
    print("\tx_true: {}".format(x_true))
    print("\td: {}".format(d_orig))
    d = d_orig.copy()
    n_true = len(x_true)
    if d['inds'] and max(d['inds'])+1 == n_true:
        n = None
    else:
        n = n_true
    print("\tn: {}".format(n))
    x = decompress_vector(d, n)
    print("\t=> x[:{}]: {}".format(len(x), x))
    assert type(x) is list and len(x) == n_true, "Output vector has the wrong length."
    assert all([abs(x_i - x_true_i) < n_true*1e-15 for x_i, x_true_i in zip(x, x_true
)])
    assert d == d_orig

# Test 1: Example
d = {}
d['inds'] = [0,   3,   7,   3,   3,   5, 1]
d['vals'] = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
x_true = [1.0, 7.0, 0.0, 11.0, 0.0, 6.0, 0.0, 3.0]
check_decompress_vector(d, x_true)

# Test 2: Random vectors
def gen_cvec_reps(p_nz, n_max):
    from random import random, randrange, sample
    x_true = [0.0] * n_max
    d = {'inds': [], 'vals': []}
    for i in range(n_max):
        if random() <= p_nz: # Create non-zero
            n_rep = randrange(1, 5)
            d['inds'].extend([i] * n_rep)
            v_i = [float("{:.2f}".format(random())) for _ in range(n_rep)]
            d['vals'].extend(v_i)
            x_true[i] = sum(v_i)
    perm = sample(range(len(d['inds'])), k=len(d['inds']))
    d['inds'] = [d['inds'][k] for k in perm]
    d['vals'] = [d['vals'][k] for k in perm]
    return (d, x_true)

p_nz = 0.2 # probability of a non-zero
n_max = 10 # maximum full-vector length
for _ in range(5): # 5 trials
    print("")
    (d, x_true) = gen_cvec_reps(p_nz, n_max)
    check_decompress_vector(d, x_true)

# Test 3: Empty vector of length 5
print("")
check_decompress_vector({'inds': [], 'vals': []}, [0.0] * 5)

print("\n(Passed!)")
```

```
Testing `decompress_vector(d, n)`:
        x_true: [1.0, 7.0, 0.0, 11.0, 0.0, 6.0, 0.0, 3.0]
        d: {'vals': [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0], 'inds': [0, 3, 7, 3, 3, 5, 1]}
        n: None
        => x[:8]: [1.0, 7.0, 0.0, 11.0, 0.0, 6.0, 0.0, 3.0]

Testing `decompress_vector(d, n)`:
        x_true: [0.96, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.06, 0.0]
        d: {'vals': [0.93, 0.96, 0.06, 0.07], 'inds': [8, 0, 8, 8]}
        n: 10
        => x[:10]: [0.96, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.06, 0.0]
```

```
Testing `decompress_vector(d, n)`:
        x_true: [0.0, 0.08, 2.02, 0.0, 0.0, 1.05, 0.4, 0.0, 0.0, 0.0]
        d: {'vals': [0.4, 0.33, 0.45, 0.9, 0.79, 0.6, 0.08], 'inds': [6, 2, 5, 2, 2, 5,
1]}
        n: 10
        => x[:10]: [0.0, 0.08, 2.02, 0.0, 0.0, 1.05, 0.4, 0.0, 0.0, 0.0]

Testing `decompress_vector(d, n)`:
        x_true: [0.0, 0.76, 0.0, 0.0, 0.0, 0.0, 1.2, 0.0, 0.0, 0.0]
        d: {'vals': [0.3, 0.76, 0.29, 0.61], 'inds': [6, 1, 6, 6]}
        n: 10
        => x[:10]: [0.0, 0.76, 0.0, 0.0, 0.0, 0.0, 1.2, 0.0, 0.0, 0.0]

Testing `decompress_vector(d, n)`:
        x_true: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.02, 0.71]
        d: {'vals': [0.53, 0.18, 0.31, 0.71], 'inds': [8, 8, 8, 9]}
        n: None
        => x[:10]: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.02, 0.71]

Testing `decompress_vector(d, n)`:
        x_true: [0.0, 0.0, 0.0, 0.0, 0.4600000000000001, 1.2199999999999998, 0.0, 0.0, 0.
0, 0.0]
        d: {'vals': [0.58, 0.33, 0.1, 0.06, 0.58, 0.03], 'inds': [5, 4, 4, 5, 5, 4]}
        n: 10
        => x[:10]: [0.0, 0.0, 0.0, 0.0, 0.4600000000000001, 1.2199999999999998, 0.0, 0.0,
0.0, 0.0]

Testing `decompress_vector(d, n)`:
        x_true: [0.0, 0.0, 0.0, 0.0, 0.0]
        d: {'vals': [], 'inds': []}
        n: 5
        => x[:5]: [0.0, 0.0, 0.0, 0.0, 0.0]

(Passed!)
```

Exercise 4 (`find_common_inds_test`: 1 point). Suppose you are given two compressed vectors, `d1` and `d2`, each represented as described above and possibly with repeated indices. Complete the function `find_common_inds(d1, d2)` so that it returns a list of the indices they have in common.

For instance, suppose:

```
d1 == {'inds': [9, 9, 1, 9, 8, 1], 'vals': [0.28, 0.84, 0.71, 0.03, 0.04, 0.75]}
    d2 == {'inds': [0, 9, 9, 1, 3, 3, 9], 'vals': [0.26, 0.06, 0.46, 0.58, 0.42, 0.21, 0.53, 0.76
]}
```

Then:

```
find_common_inds(d1, d2) == [1, 9]
```

> **Note 1.** The returned list must not have duplicate indices, even if the inputs do. In the example, the index 9 is repeated in both `d1` and `d2`, but the output includes just one 9.
>
> **Note 2.** In the returned list, the order of indices does not matter. For instance, the example shows `[1, 9]` but `[9, 1]` would also be valid.

In [9]:

Student's answer (Top)

```
def find_common_inds(d1, d2):
    assert type(d1) is dict and 'inds' in d1 and 'vals' in d1
    assert type(d2) is dict and 'inds' in d2 and 'vals' in d2
    s1 = set(d1['inds'])
    s2 = set(d2['inds'])
    return list(s1 & s2)
```

In [10]:

Grade cell: `find_common_inds_test` Score: 1.0 / 1.0 (Top)

```
# `find_common_inds_test`: Test cell
def check_find_common_inds(d1, d2, ans):
    print("Testing `check_find_common_inds(d1, d2, ans)`:")
    print("\td1: {}".format(d1))
    print("\td2: {}".format(d2))
    print("\texpected ans: {}".format(ans))
    common = find_common_inds(d1, d2)
    print("\tcomputed common: {}".format(common))
    assert type(common) is list
    assert sorted(common) == sorted(ans), "Answers do not match."

# Test 1: Example
d1 = {'inds': [9, 9, 1, 9, 8, 1], 'vals': [0.28, 0.84, 0.71, 0.03, 0.04, 0.75]}
d2 = {'inds': [0, 9, 9, 1, 3, 3, 9], 'vals': [0.26, 0.06, 0.46, 0.58, 0.42, 0.21, 0.53,
0.76]}
```

```
0.76]}
ans = [1, 9]
check_find_common_inds(d1, d2, ans)

# Test 2: Random tests
from random import random, randrange, sample, shuffle
p_common = 0.2
for _ in range(5):
    print("")
    n_min = 10
    x = sample(range(2*n_min), 2*n_min)
    i1, i2 = x[:n_min], x[n_min:]
    inds1, inds2 = [], []
    ans = []
    for k, i in enumerate(i1):
        if random() <= p_common:
            i2[k] = i
            ans.append(i)
        inds1.extend([i] * randrange(1, 4))
        inds2.extend([i2[k]] * randrange(1, 4))
    shuffle(inds1)
    d1 = {'inds': inds1, 'vals': [float("{:.1f}".format(random())) for _ in range(len(i
nds1))]}
    shuffle(inds2)
    d2 = {'inds': inds2, 'vals': [float("{:.1f}".format(random())) for _ in range(len(i
nds2))]}
    check_find_common_inds(d1, d2, ans)

print("\n(Passed!)")
```

```
Testing `check_find_common_inds(d1, d2, ans)`:
        d1: {'vals': [0.28, 0.84, 0.71, 0.03, 0.04, 0.75], 'inds': [9, 9, 1, 9, 8, 1]}
        d2: {'vals': [0.26, 0.06, 0.46, 0.58, 0.42, 0.21, 0.53, 0.76], 'inds': [0, 9, 9,
1, 3, 3, 9]}
        expected ans: [1, 9]
        computed common: [9, 1]

Testing `check_find_common_inds(d1, d2, ans)`:
        d1: {'vals': [0.8, 0.8, 0.9, 0.8, 0.6, 0.2, 0.8, 0.5, 0.5, 0.1, 0.9, 0.8, 0.1, 0.
3, 0.6, 0.2, 0.8, 0.9], 'inds': [7, 3, 4, 8, 19, 13, 18, 18, 2, 3, 19, 7, 10, 10, 10, 9,
13, 2]}
        d2: {'vals': [0.7, 0.3, 0.2, 0.8, 0.1, 0.9, 0.7, 0.1, 0.6, 0.3, 0.0, 0.4, 0.9, 0.
4, 0.3, 0.7, 0.2, 1.0, 0.4, 0.5, 0.4, 0.3, 0.5, 0.9], 'inds': [16, 18, 2, 18, 16, 2, 11,
0, 3, 3, 16, 6, 17, 11, 14, 14, 11, 2, 0, 3, 5, 18, 17, 14]}
        expected ans: [3, 18, 2]
        computed common: [18, 2, 3]

Testing `check_find_common_inds(d1, d2, ans)`:
        d1: {'vals': [0.7, 0.8, 0.7, 0.7, 0.6, 0.8, 1.0, 0.3, 0.2, 0.8, 0.6, 0.4, 0.5, 0.
0, 0.5, 0.7, 1.0, 0.6, 0.6, 0.6, 0.5, 0.6], 'inds': [3, 0, 0, 14, 5, 18, 16, 11, 10, 14,
4, 1, 0, 18, 1, 11, 18, 16, 10, 11, 16, 1]}
        d2: {'vals': [0.4, 0.8, 0.5, 0.9, 0.6, 0.4, 0.9, 0.5, 0.4, 0.4, 0.4, 0.1, 0.4, 0.
2, 0.2, 0.1, 0.0, 0.2, 0.7, 0.0, 0.9], 'inds': [18, 18, 12, 9, 8, 18, 13, 9, 12, 7, 2, 1
5, 12, 4, 13, 13, 17, 2, 4, 7, 2]}
        expected ans: [4, 18]
        computed common: [18, 4]

Testing `check_find_common_inds(d1, d2, ans)`:
        d1: {'vals': [0.9, 0.0, 0.7, 0.7, 0.3, 1.0, 0.0, 0.1, 0.6, 0.4, 0.9, 0.6, 0.1, 0.
1, 0.7, 0.5, 0.0, 0.2, 0.3, 0.9, 0.8], 'inds': [17, 15, 4, 18, 5, 1, 15, 15, 18, 19, 14,
14, 16, 16, 4, 12, 1, 1, 12, 19, 4]}
        d2: {'vals': [0.7, 0.5, 0.2, 0.1, 0.9, 0.4, 0.6, 0.6, 0.4, 0.1, 0.9, 0.4, 0.6, 0.
4, 0.9, 0.4, 0.2, 0.0, 0.8, 0.5, 0.7], 'inds': [3, 0, 13, 3, 2, 19, 19, 6, 6, 10, 7, 13,
0, 0, 10, 3, 2, 9, 6, 19, 11]}
        expected ans: [19]
        computed common: [19]

Testing `check_find_common_inds(d1, d2, ans)`:
        d1: {'vals': [0.8, 0.6, 0.5, 1.0, 0.5, 0.4, 0.6, 0.3, 0.8, 1.0, 1.0, 0.0, 0.7, 0.
8, 0.0, 0.7, 0.3, 0.6, 0.2, 0.5, 0.6, 0.0, 0.9, 0.6], 'inds': [12, 8, 1, 14, 10, 11, 14,
12, 11, 6, 19, 11, 8, 1, 6, 14, 19, 0, 8, 19, 0, 9, 12, 10]}
        d2: {'vals': [0.2, 0.1, 0.8, 1.0, 0.8, 0.3, 1.0, 0.1, 0.5, 0.4, 0.4, 0.1, 0.3, 0.
7, 0.4, 0.0, 0.0], 'inds': [16, 8, 7, 13, 7, 5, 15, 13, 18, 16, 2, 5, 5, 3, 16, 18, 4]}
        expected ans: [8]
        computed common: [8]

Testing `check_find_common_inds(d1, d2, ans)`:
        d1: {'vals': [0.3, 0.5, 0.2, 1.0, 0.3, 0.8, 0.5, 0.2, 0.1, 0.4, 0.3, 0.9, 0.8, 0.
1, 0.1, 0.2, 0.3, 0.5], 'inds': [2, 9, 10, 2, 10, 14, 7, 2, 0, 7, 5, 18, 18, 3, 14, 15,
7, 18]}
        d2: {'vals': [0.5, 1.0, 0.4, 0.8, 0.9, 0.4, 0.1, 0.1, 0.7, 0.1, 0.2, 0.7, 0.5, 0.
1, 0.9, 0.7, 0.0, 0.9, 0.3], 'inds': [8, 3, 13, 11, 18, 11, 16, 6, 12, 3, 2, 13, 2, 18, 1
6, 6, 8, 6, 0]}
        expected ans: [3, 2, 18, 0]
        computed common: [0, 18, 2, 3]

(Passed!))
```

In [11]:

2-more_exercises (Score: 11.0 / 11.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 1.0 / 1.0)
4. Test cell (Score: 2.0 / 2.0)
5. Test cell (Score: 1.0 / 1.0)
6. Test cell (Score: 2.0 / 2.0)
7. Test cell (Score: 1.0 / 1.0)
8. Test cell (Score: 2.0 / 2.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Python review: More exercises

This notebook continues the review of Python basics based on Chris Simpkins's (https://www.cc.gatech.edu/~simpkins/) Python Bootcamp (https://www.cc.gatech.edu/~simpkins/teaching/python-bootcamp/syllabus.html).

This particular notebook adapts the exercises that appeared with the "Functional Programming" slides (https://www.cc.gatech.edu/~simpkins/teaching/python-bootcamp/slides/functional-programming.html) of the Fall 2016 offering.

Consider the following dataset of exam grades, organized as a 2-D table and stored in Python as a "list of lists" under the variable name, `grades`.

In [1]:
```
grades = [
    # First line is descriptive header. Subsequent lines hold data
    ['Student', 'Exam 1', 'Exam 2', 'Exam 3'],
    ['Thorny', '100', '90', '80'],
    ['Mac', '88', '99', '111'],
    ['Farva', '45', '56', '67'],
    ['Rabbit', '59', '61', '67'],
    ['Ursula', '73', '79', '83'],
    ['Foster', '89', '97', '101']
]
```

Exercise 0 (`students_test`: 1 point). Write some code that computes a new list named `students[:]`, which holds the names of the students as they from "top to bottom" in the table.

In [2]:
Student's answer (Top)

```
students = [L[0] for L in grades[1:]]
```

In [3]:
Grade cell: `students_test` Score: 1.0 / 1.0 (Top)

```
# `students_test`: Test cell
print(students)
assert type(students) is list
```

```
assert students == ['Thorny', 'Mac', 'Farva', 'Rabbit', 'Ursula', 'Foster']
print("\n(Passed!)")
```

```
['Thorny', 'Mac', 'Farva', 'Rabbit', 'Ursula', 'Foster']

(Passed!)
```

Exercise 1 (`assignments_test`: 1 point). Write some code to compute a new list named `assignments[:]`, to hold the names of the class assignments. (These appear in the descriptive header element of `grades`.)

In [4]:

Student's answer (Top)

```
assignments = grades[0][1:]
```

In [5]:

Grade cell: `assignments_test` Score: 1.0 / 1.0 (Top)

```
# `assignments_test`: Test cell
print(assignments)
assert type(assignments) is list
assert assignments == ['Exam 1', 'Exam 2', 'Exam 3']
print("\n(Passed!)")
```

```
['Exam 1', 'Exam 2', 'Exam 3']

(Passed!)
```

Exercise 2 (`grade_lists_test`: 1 point). Write some code to compute a new *dictionary*, named `grade_lists`, that maps names of students to *lists* of their exam grades. The grades should be converted from strings to integers. For instance, `grade_lists['Thorny'] == [100, 90, 80]`.

In [6]:

Student's answer (Top)

```
# Create a dict mapping names to lists of grades.

# One-line solution: It works, and is vaguely clever, but it is not pretty.
#grade_lists = {L[0]: [int(g) for g in L[1:]] for L in grades[1:]}

# Alternative: More verbose but (arguably) more readable
grade_lists = {} # Empty dictionary
for L in grades[1:]:
    grade_lists[L[0]] = [int(g) for g in L[1:]]
```

In [7]:

Grade cell: `grade_lists_test` Score: 1.0 / 1.0 (Top)

```
# `grade_lists_test`: Test cell
print(grade_lists)
assert type(grade_lists) is dict, "Did not create a dictionary."
assert len(grade_lists) == len(grades)-1, "Dictionary has the wrong number of entries."
assert {'Thorny', 'Mac', 'Farva', 'Rabbit', 'Ursula', 'Foster'} == set(grade_lists.keys
()), "Dictionary has the wrong keys."
assert grade_lists['Thorny'] == [100, 90, 80], 'Wrong grades for: Thorny'
assert grade_lists['Mac'] == [88, 99, 111], 'Wrong grades for: Mac'
assert grade_lists['Farva'] == [45, 56, 67], 'Wrong grades for: Farva'
assert grade_lists['Rabbit'] == [59, 61, 67], 'Wrong grades for: Rabbit'
assert grade_lists['Ursula'] == [73, 79, 83], 'Wrong grades for: Ursula'
assert grade_lists['Foster'] == [89, 97, 101], 'Wrong grades for: Foster'
print("\n(Passed!)")
```

```
{'Foster': [89, 97, 101], 'Thorny': [100, 90, 80], 'Ursula': [73, 79, 83], 'Farva': [45,
56, 67], 'Mac': [88, 99, 111], 'Rabbit': [59, 61, 67]}

(Passed!)
```

Exercise 3 (`grade_dicts_test`: 2 points). Write some code to compute a new dictionary, `grade_dicts`, that maps names of students to *dictionaries* containing their scores. Each entry of this scores dictionary should be keyed on assignment name and hold the corresponding grade as an integer. For instance, `grade_dicts['Thorny']['Exam 1'] == 100`.

In [8]:

Student's answer (Top)

```
# Create a dict mapping names to dictionaries of grades.

# One-line solution: It works, and is vaguely clever, but it is not pretty.
#grade_dicts = {L[0]: dict(zip(assignments, [int(g) for g in L[1:]])) for L in grades
```

```
[1:]}

# Alternative: More verbose but (arguably) more readable
grade_dicts = {} # Empty
for L in grades[1:]:
    grade_dicts[L[0]] = dict(zip(assignments, [int(g) for g in L[1:]]))
```

In [9]:

Grade cell: `grade_dicts_test` Score: 2.0 / 2.0 (Top)

```
# `grade_dicts_test`: Test cell
print(grade_dicts)
assert type(grade_dicts) is dict, "Did not create a dictionary."
assert len(grade_dicts) == len(grades)-1, "Dictionary has the wrong number of entries."
assert {'Thorny', 'Mac', 'Farva', 'Rabbit', 'Ursula', 'Foster'} == set(grade_dicts.keys
()), "Dictionary has the wrong keys."
assert grade_dicts['Foster']['Exam 1'] == 89, 'Wrong score'
assert grade_dicts['Foster']['Exam 3'] == 101, 'Wrong score'
assert grade_dicts['Foster']['Exam 2'] == 97, 'Wrong score'
assert grade_dicts['Ursula']['Exam 1'] == 73, 'Wrong score'
assert grade_dicts['Ursula']['Exam 3'] == 83, 'Wrong score'
assert grade_dicts['Ursula']['Exam 2'] == 79, 'Wrong score'
assert grade_dicts['Rabbit']['Exam 1'] == 59, 'Wrong score'
assert grade_dicts['Rabbit']['Exam 3'] == 67, 'Wrong score'
assert grade_dicts['Rabbit']['Exam 2'] == 61, 'Wrong score'
assert grade_dicts['Mac']['Exam 1'] == 88, 'Wrong score'
assert grade_dicts['Mac']['Exam 3'] == 111, 'Wrong score'
assert grade_dicts['Mac']['Exam 2'] == 99, 'Wrong score'
assert grade_dicts['Farva']['Exam 1'] == 45, 'Wrong score'
assert grade_dicts['Farva']['Exam 3'] == 67, 'Wrong score'
assert grade_dicts['Farva']['Exam 2'] == 56, 'Wrong score'
assert grade_dicts['Thorny']['Exam 1'] == 100, 'Wrong score'
assert grade_dicts['Thorny']['Exam 3'] == 80, 'Wrong score'
assert grade_dicts['Thorny']['Exam 2'] == 90, 'Wrong score'
print("\n(Passed!)")
```

```
{'Foster': {'Exam 1': 89, 'Exam 3': 101, 'Exam 2': 97}, 'Thorny': {'Exam 1': 100, 'Exam
3': 80, 'Exam 2': 90}, 'Ursula': {'Exam 1': 73, 'Exam 3': 83, 'Exam 2': 79}, 'Farva': {'E
xam 1': 45, 'Exam 3': 67, 'Exam 2': 56}, 'Mac': {'Exam 1': 88, 'Exam 3': 111, 'Exam 2': 9
9}, 'Rabbit': {'Exam 1': 59, 'Exam 3': 67, 'Exam 2': 61}}

(Passed!)
```

Exercise 4 (`avg_grades_by_student_test`: 1 point). Write some code to compute a dictionary named `avg_grades_by_student` that maps each student to his or her average exam score. For instance, `avg_grades_by_student['Thorny'] == 90`.

> **Hint.** The `statistics` (https://docs.python.org/3.5/library/statistics.html) module of Python has at least one helpful function.

In [10]:

Student's answer (Top)

```
# Create a dict mapping names to grade averages.
from statistics import mean
avg_grades_by_student = {n: mean(G) for n, G in grade_lists.items()}
```

In [11]:

Grade cell: `avg_grades_by_student_test` Score: 1.0 / 1.0 (Top)

```
# `avg_grades_by_student_test`: Test cell
print(avg_grades_by_student)
assert type(avg_grades_by_student) is dict, "Did not create a dictionary."
assert len(avg_grades_by_student) == len(students), "Output has the wrong number of stu
dents."
assert abs(avg_grades_by_student['Mac'] - 99.33333333333333) <= 4e-15, 'Mean is incorre
ct'
assert abs(avg_grades_by_student['Foster'] - 95.66666666666667) <= 4e-15, 'Mean is inco
rrect'
assert abs(avg_grades_by_student['Farva'] - 56) <= 4e-15, 'Mean is incorrect'
assert abs(avg_grades_by_student['Rabbit'] - 62.333333333333336) <= 4e-15, 'Mean is inc
orrect'
assert abs(avg_grades_by_student['Thorny'] - 90) <= 4e-15, 'Mean is incorrect'
assert abs(avg_grades_by_student['Ursula'] - 78.33333333333333) <= 4e-15, 'Mean is inco
rrect'
print("\n(Passed!)")
```

```
{'Foster': 95.66666666666667, 'Thorny': 90, 'Ursula': 78.33333333333333, 'Farva': 56, 'Ma
c': 99.33333333333333, 'Rabbit': 62.333333333333336}

(Passed!)
```

Exercise 5 (`grades_by_assignment_test`: 2 points). Write some code to compute a dictionary named `grades_by_assignment`, whose keys are assignment (exam) names and whose values are lists of scores over all students on that assignment. For instance, `grades_by_assignment['Exam 1'] == [100, 88, 45, 59, 73, 89]`.

In [12]:

Student's answer (Top)

```
# One-line solution: It works, and is vaguely clever, but it is not pretty.
#grades_by_assignment = {a: [int(L[k]) for L in grades[1:]] for k, a in zip(range(1,
 4), assignments)}

# Alternative: More verbose but (arguably) more readable
grades_by_assignment = {} # Empty dictionary
for k, a in enumerate(assignments): # (0, 'Exam 1'), ...
    grades_by_assignment[a] = [int(L[k+1]) for L in grades[1:]]
```

In [13]:

Grade cell: `grades_by_assignment_test` Score: 2.0 / 2.0 (Top)

```
# `grades_by_assignment_test`: Test cell
print(grades_by_assignment)
assert type(grades_by_assignment) is dict, "Output is not a dictionary."
assert len(grades_by_assignment) == 3, "Wrong number of assignments."
assert grades_by_assignment['Exam 1'] == [100, 88, 45, 59, 73, 89], 'Wrong grades list'
assert grades_by_assignment['Exam 3'] == [80, 111, 67, 67, 83, 101], 'Wrong grades lis
t'
assert grades_by_assignment['Exam 2'] == [90, 99, 56, 61, 79, 97], 'Wrong grades list'
print("\n(Passed!)")
```

```
{'Exam 1': [100, 88, 45, 59, 73, 89], 'Exam 3': [80, 111, 67, 67, 83, 101], 'Exam 2': [9
0, 99, 56, 61, 79, 97]}

(Passed!)
```

Exercise 6 (`avg_grades_by_assignment_test`: 1 point). Write some code to compute a dictionary, `avg_grades_by_assignment`, which maps each exam to its average score.

In [14]:

Student's answer (Top)

```
# Create a dict mapping items to average for that item across all students.
from statistics import mean
avg_grades_by_assignment = {a: mean(G) for a, G in grades_by_assignment.items()}
```

In [15]:

Grade cell: `avg_grades_by_assignment_test` Score: 1.0 / 1.0 (Top)

```
# `avg_grades_by_assignment_test`: Test cell
print(avg_grades_by_assignment)
assert type(avg_grades_by_assignment) is dict
assert len(avg_grades_by_assignment) == 3
assert abs((100+88+45+59+73+89)/6 - avg_grades_by_assignment['Exam 1']) <= 7e-15
assert abs((80+111+67+67+83+101)/6 - avg_grades_by_assignment['Exam 3']) <= 7e-15
assert abs((90+99+56+61+79+97)/6 - avg_grades_by_assignment['Exam 2']) <= 7e-15
print("\n(Passed!)")
```

```
{'Exam 1': 75.66666666666667, 'Exam 3': 84.83333333333333, 'Exam 2': 80.33333333333333}

(Passed!)
```

Exercise 7 (`rank_test`: 2 points). Write some code to create a new list, `rank`, which contains the names of students in order by *decreasing* score. That is, `rank[0]` should contain the name of the top student (highest average exam score), and `rank[-1]` should have the name of the bottom student (lowest average exam score).

In [16]:

Student's answer (Top)

```
rank = sorted(avg_grades_by_student, key=avg_grades_by_student.get, reverse=True)
```

In [17]:

Grade cell: `rank_test` Score: 2.0 / 2.0 (Top)

```
# `rank_test`: Test cell
print(rank)
print("\n=== Ranking ===")
for i, s in enumerate(rank):
    print("{}. {}: {}".format(i+1, s, avg_grades_by_student[s]))
```

```
assert rank == ['Mac', 'Foster', 'Thorny', 'Ursula', 'Rabbit', 'Farva']
for i in range(len(rank)-1):
    assert avg_grades_by_student[rank[i]] >= avg_grades_by_student[rank[i+1]]
print("\n(Passed!)")
```

```
['Mac', 'Foster', 'Thorny', 'Ursula', 'Rabbit', 'Farva']

=== Ranking ===
1. Mac: 99.33333333333333
2. Foster: 95.66666666666667
3. Thorny: 90
4. Ursula: 78.33333333333333
5. Rabbit: 62.333333333333336
6. Farva: 56

(Passed!)
```

In [18]: