ICS 31 • UC Irvine
Fall 2012 • David G. Kay

Your Name _____ Your lab:
Your Student ID (8 digits) _____ Section (1-10) _____
Your UCInet ID _____ Time MWF at: 8 10 12 2 4 6
TA's name _____

Second Midterm

You have 75 minutes (until the end of the class period) to complete this exam. There are 55 points possible, so allow approximately one minute per point and you'll have plenty of time left over.

Please read all the problems carefully. If you have a question on what a problem means or what it calls for, ask us. Unless a problem specifically asks about errors, you should assume that each problem is correct and solvable; ask us if you believe otherwise.

In answering these questions, you may use any Python 3 features we have covered in class, in the text, in the lab assignments, or earlier on the exam, unless a problem says otherwise. Use more advanced features at your own risk; you must use them correctly.

Remember, stay cool! If you run into trouble on a problem, go on to the next one. Later on, you can go back if you have time. Don't let yourself get stuck on any one problem.

You may not share any information or materials with classmates during the exam and you may not use any electronic devices.

Please write your answers clearly and neatly—we can't give you credit if we can't decipher what you've written.

We'll give partial credit for partially correct answers, so writing something is better than writing nothing. But be sure to answer just what the question asks.

Good luck!

Problem 1 (7 points)
Problem 2 (10 points)
Problem 3 (3 points)
Problem 4 (9 points)
Problem 5 p6 (6 points)
Problem 5 p7 (10 points)
Problem 5 p8 (10 points)
Total (55 points)

aJm34w

Problem 1 (7 points) **Topic: Python expressions and data types**

Below are seven segments of code, each with a part underlined. Indicate the data type of each underlined part by checking the appropriate box.

(a) ❑ int ❑ float ❑ bool ❑ str ❑ function ❑ list of int ❑ list of str

```
if temp < 32:                          # bool
    print("Freezing")
```

(b) ❑ int ❑ float ❑ bool ❑ str ❑ function ❑ list of int ❑ list of str

```
L = ['Huey', 'Dewey', 'Louie']         # str
print(L[1])
```

(c) ❑ int ❑ float ❑ bool ❑ str ❑ function ❑ list of int ❑ list of str

```
M = [ ]                                # list of int
for i in range(3):
    M.append(i)
    print(M)
```

(d) ❑ int ❑ float ❑ bool ❑ str ❑ function ❑ list of int ❑ list of str

```
L = ['Huey', 'Dewey', 'Louie']         # int
n = len(L)
if 'Donald' in L[1:n]:
   print(L)
```

(e) ❑ int ❑ float ❑ bool ❑ str ❑ function ❑ list of int ❑ list of str

```
def is_even(n: int) -> bool:           # bool
    return n % 2 == 0
```

(f) ❑ int ❑ float ❑ bool ❑ str ❑ function ❑ list of int ❑ list of str

```
L = inputline.split()                  # list of str
while L != [ ]:
    print(L)
    L = L[1:]
```

(g) ❑ int ❑ float ❑ bool ❑ str ❑ function ❑ list of int ❑ list of str

```
L = ['Huey', 'Dewey', 'Louie']         # str
print(L[0] + L[1])
```

SCORING: 1 point each

Problem 2 (10 points) **Topic: Loop behavior**

For this problem, use these definitions:

```
L = ['BIM', 'CGS', 'CS', 'CSE', 'Infx']
M = [1992, 1996, 2008, 2012]
N = [2, 3, 5]
```

Match each of the following code segments ((a) through (e)) with the results (A through H) they produce when run in Python. You may use some results (A through H) more than once.

(a) Circle one: A B C D E F G H **---> B**

```
for v in range(len(M)):
    print(v, M[v], len(M))
print('Done', len(M))
```

(b) Circle one: A B C D E F G H **---> H**

```
for v in range(5):
    print(v, L[v], len(L))
print('Done', len(L))
```

(c) Circle one: A B C D E F G H **---> C**

```
for v in L:
    print(v, L[v], len(L))
print('Done', len(L))
```

(d) Circle one: A B C D E F G H **---> E**

```
for v in N:
    print(v, 2 * v, len(N))
print('Done', len(N))
```

(e) Circle one: A B C D E F G H **---> B**

```
v = 0
while v < len(M):
    print(v, M[v], len(M))
    v = v + 1
print('Done', len(M))
```

SCORING: 2 points each

A.

```
0 1992 4 Done 4
1 1996 4 Done 4
2 2008 4 Done 4
3 2012 4 Done 4
```

B.

```
0 1992 4
1 1996 4
2 2008 4
3 2012 4
Done 4
```

C.

```
TypeError: list indices
must be integers, not str
```

D.

```
1 BIM 0
2 CGS 1
3 CS 2
4 CSE 3
5 Infx 4
Done 5
```

E.

```
2 4 3
3 6 3
5 10 3
Done 3
```

F.

```
1 4 3
2 6 3
3 10 3
Done 3
```

G.

```
0 1992 0
1 1996 1
2 2008 2
3 2012 3
Done 4
```

H.

```
0 BIM 5
1 CGS 5
2 CS 5
3 CSE 5
4 Infx 5
Done 5
```

Problem 3 (3 points) **Topic: String formatting**

The California Secretary of State posts the results of ballot measures on her web site in a form much like this:

```
Yes   30  Temporary Taxes to Fund Education            5457850  54.2%  4620176  45.8%
No    31  State Budget, State and Local Government     3692410  39.3%  5702549  60.7%
```

We could represent this data as follows:

```
Results = namedtuple('Results', 'num title yes no')
# Num is an integer, the measure number; title is a string;
# yes is the number of yes votes; no is the number of no votes
firstprop = Results(30, 'Temporary Taxes to Fund Education', 5457850, 4620176)
secondprop= Results(31, 'State Budget, State and Local Government', 3692410, 5702549)
```

The function below should print the results as shown above when the correct format string is inserted:

```
def print_results_row(R: Results) -> None:
    ''' Print one line of election results for a ballot measure (see format above).
    '''
    format_string = — Choose one from the five format string (A–E) shown below —
    if R.yes > R.no:
        outcome = 'Yes'
    else:
        outcome = 'No'
    total_votes = R.yes + R.no
    print(format_string.format(outcome, R.num, R.title, R.yes, R.yes/total_votes*100,
        R.no, R.no/total_votes*100))
    return

print_results_row(firstprop)
print_results_row(secondprop)
```

Choose the one format string below (A through E) that most correctly produces the output shown above.

A. `"{:3}  {:2d}  {:45}{:8d} {:5.4f}% {:8d} {:5.4f}%"`

B. `"{}  {:2d}  {}{:8d} {:5.1f}% {:8d} {:5.1f}%"`

C. `"{:3}  {:2d}  {:45}{:8d} {:5.1f}% {:8d} {:5.1f}%"` **<— THIS ONE**

D. `"{:3}  {:2d}  {:>45}{:8d} {:5.2f}% {:8d} {:5.2f}%"`

E. `"{:3}  {:2d}  {:45}{:8d} {:8.1f}% {:8d} {:8.1f}%"`

SCORING: Three points for the right answer

Problem 4 (9 points) **Topic: if/elif/else behavior**

The fee to insure an Express Mail shipment to Denmark is calculated based on the value of the shipment, according to this table:

$100 and below, no fee
over $100 but $200 and below, $0.85
over $200 but $500 and below, $2.35
over $500 but $650 and below, $3.85
over $650, no fee because the package is not insurable

Here is the framework for a function to calculate the insurance fee based on the shipment's value; the body of the function is missing.

```
def insurance_fee(value: float) -> float:
    ''' Return insurance fee based on package value as described above.
        Print message if value is too high to insure.
    '''
    — Insert body of function here —
```

```
assert(insurance_fee(100) == 0)
assert(insurance_fee(101) == 0.85)
assert(insurance_fee(200) == 0.85)
assert(insurance_fee(201) == 2.35)
assert(insurance_fee(500) == 2.35)
assert(insurance_fee(501) == 3.85)
assert(insurance_fee(650) == 3.85)
assert(insurance_fee(651) == 0) # and a message is printed in this case
```

Below are six alternatives for the body of this function. One or more of them may correctly satisfy the specifications. Indicate which of the six alternatives is correct by circling *one or more* of the following:

A B C D E F **SCORING: Start with 9 pts. –1.5 for each wrong mark (yes for no, no for yes)**

A. Correct

```
if value <= 100:
   fee = 0
elif value <= 200:
   fee = 0.85
elif value <= 500:
   fee = 2.35
elif value <= 650:
   fee = 3.85
else:
   fee = 0
   print('Not insurable')
return fee
```

B. Wrong

```
if value <= 100:
    return 0
elif value <= 200:
    return 0.85
elif value <= 500:
    return 2.35
elif value <= 650:
    return 3.85
else:
    return 0
    print('Not insurable')
```

C. Correct

```
if value <= 100:
    return 0
elif value <= 200:
    return 0.85
elif value <= 500:
    return 2.35
elif value <= 650:
    return 3.85
else:
    print('Not insurable')
    return 0
```

D. Correct

```
if value<=100:
    fee = 0
if value>100 and value<=200:
    fee = 0.85
if value>200 and value<=500:
    fee = 2.35
if value>500 and value<=650:
    fee = 3.85
if value > 650:
    fee = 0
    print('Not insurable')
return fee
```

E. Wrong

```
if value <= 100:
    fee = 0
if value <= 200:
    fee = 0.85
if value <= 500:
    fee = 2.35
if value <= 650:
    fee = 3.85
if value > 650:
    fee = 0
    print('Not insurable')
return fee
```

F. Correct

```
fee = 0
if value>100 and value<=200:
    fee = 0.85
elif value>200 and value<=500:
    fee = 2.35
elif value>500 and value<=650:
    fee = 3.85
elif value>650:
    print('Not insurable')
return fee
```

Problem 5 (26 points) **Topic: Processing lists and namedtuples**

For this problem, use these definitions:

```
from collections import namedtuple
Restaurant = namedtuple('Restaurant', 'name cuisine phone menu')
Dish = namedtuple('Dish', 'name price calories')
```

The menu field of a Restaurant is a list of Dish structures.

You may also find this excerpt from `help(str)` useful:

```
count(...)
    S.count(sub) -> int
    Return the number of non-overlapping occurrences of substring sub in
    string S.

endswith(...)
    S.endswith(suffix) -> bool
    Return True if S ends with the specified suffix, False otherwise.

find(...)
    S.find(sub) -> int
    Return the lowest index in S where substring sub is found.
    Return -1 on failure.

startswith(...)
    S.startswith(prefix) -> bool
    Return True if S starts with the specified prefix, False otherwise.
```

(a) (2 points) Complete the definition of the function below, consistent with its header, docstring comment, and assertions.

```
def Dish_name_is(a_dish: Dish, n: str) -> bool:
    ''' Return True if a_dish's name equals n (and False otherwise) '''
    return a_dish.name == n


assert(Dish_name_is(Dish("Doro Wat", 12.50, 550), "Doro Wat"))
assert(not Dish_name_is(Dish("Doro Wat", 12.50, 550), "Doro"))
```

(b) (4 points) Complete the definition of the function below, consistent with its header, docstring comment, and assertions.

```
def Dish_name_contains(a_dish: Dish, a_phrase: str) -> bool:
    ''' Return True if a_dish's name includes the second parameter '''
    return a_phrase in a_dish.name  —OR—  return a_dish.name.find(a_phrase) != -1
        —OR— return a_dish.name.count(a_phrase) > 0


assert(Dish_name_contains(Dish("Yesiga Tibs", 12.50, 550), "Yesiga Tibs"))
assert(Dish_name_contains(Dish("Yesiga Tibs", 12.50, 550), "Tibs"))
assert(not Dish_name_contains(Dish("Yesiga Tibs", 12.50, 550), "Doro Wat"))
assert(not Dish_name_contains(Dish("Yesiga Tibs", 12.50, 550), "YT"))
```

(c) (2 points) At the beginning of this problem, four string methods are described: `find`, `count`, `startswith`, and `endswith`. Regardless of how you answered part (b), two of these four methods *could be* used in a short definition of `Dish_name_contains`; the other two would require much more code to solve the problem. Which two of these four string methods could be used in a brief definition of `Dish_name_contains`? (Just give their names.) **The find() and count() methods could be used easily, as shown above. It would be possible to use startswith() or endswith(), but it would be much clumsier and less efficient (you'd have to do multiple searches to examine the whole string). SCORING: +1 for each correct answer (find/count); -1 for each incorrect answer (startswith, endswith, other things); minimum zero.**

(d) (6 points) Complete the definition of the function below, consistent with its header, docstring comment, and assertions. For full credit, use functions described in earlier problems on this exam where appropriate (whether or not you solved those problems correctly). Fill each blank with exactly one identifier or constant.

```
def Menu_includes (M: 'list of Dish', a_phrase: str) -> bool:
    ''' Return True if M includes at least one dish whose name includes the second
        parameter.
    '''
    for d in ____________________:
        if ____________________(____________________, ____________________):
            return ____________________
    return ____________________
```

M. Next line: Dish_name_contains … d … a_phrase

True

False. SCORING: 1 point per correct blank.

```
DL = [Dish('spaghetti and meatballs', 9.50, 600),
      Dish('cheeseburger', 7.60, 800),
      Dish('macaroni and cheese', 3.50, 600)]
assert(Menu_includes(DL, 'cheese'))
assert(not Menu_includes(DL, 'carrot'))
assert(Menu_includes(DL, 'meat'))
assert(not Menu_includes(DL, 'cheeses'))
```

(e) (2 points) Here is a version of `Menu_includes` in which the last line is indented one step to the right. This version is no longer correct.

```
def Menu_includes (M: 'list of Dish', a_phrase: str) -> bool:
    ''' Return True if M includes at least one dish whose name includes the second
        parameter.
    '''
    for d in ____________________:
        if ____________________(____________________, ____________________):
            return ____________________
        return ____________________
```

```
assert(Menu_includes(DL, 'cheese'))      #1
assert(not Menu_includes(DL, 'carrot'))  #2
assert(Menu_includes(DL, 'meat'))        #3
assert(not Menu_includes(DL, 'cheeses')) #4
```

At least one of the four assertions above will fail. Which assertion(s) fail with this mis-indented version? Just indicate the number(s) 1, 2, 3, and/or 4. **ANSWER: 'cheese' fails (number varies by version).**

(f) (4 points) Complete the definition of the function below, consistent with its header, docstring comment, and assertions. For full credit, use functions described earlier on this exam where appropriate (whether or not you solved those problems correctly). Fill each blank with exactly one identifier or constant.

```
def Restaurant_serves(R: Restaurant, a_phrase: str) -> bool:
    ''' Return True if R serves a dish whose name includes the second parameter.
    '''
    return _________________(_________________._________________, _________________)
```

return Menu_includes(R.menu, a_phrase). SCORING: 1 point per blank

(g) (6 points) Complete the definition of the function below, consistent with its header, docstring comment, and assertions. For full credit, use functions described earlier on this exam where appropriate (whether or not you solved those problems correctly). Fill each blank with exactly one identifier or constant.

```
def Restaurants_serving(RestaurantColl: 'list of Restaurant',
                        a_phrase: str) -> 'list of Restaurant':
    ''' Return a list of those restaurants in RL that serve
        a dish whose name includes the second parameter.
    '''
    result = [ ]

    for r in ____________________________:
             RestaurantColl
        if ____________________________(____________________________, a_phrase):
             Restaurant_serves                          r
            ____________________________.append(____________________________)
             result                                     r
    return ____________________________
             result
```

Scoring: 1 point per blank