1. Determine the value and type of each of the expressions below. If the question has two lines, assume that the statement in the first line is executed, and give the value and type of the expression in the second line. Note that it is possible for an expression to have type NoneType and value None, and also possible for an expression to be illegal and cause an error when it is evaluated.

(a) \(8/\div 4\)

2 int

(b) \(8/4\)

2.0 float

(c) ‘20’*3+3

error, adding a string and an int

(d) int(‘20’*3)+’3’

also an error, adding an int and a string, and an error on my part as well, because I meant to leave the quotation marks off of the 3 , which would have made 202023 (type int) the correct answer!

(e) (10,’ten’,’X’)[1:3]+(11,)

(‘ten’,’X’,11) tuple

(f) d={'alchemist':(‘a’,’t’),’shaman’: (‘s’,’n’),’wizard’: (‘w’,’z’)}
    d[‘shaman’][1]

‘n’ string

(g) x=[3,9,13]
    x.append(17)

None NoneType

(h) y=[[2,3,4],[5,6,7],[8,9,10]]
    [y[i][i] for i in range(len(y))]

None NoneType
Most students did very well with this problem. Issues were occasionally reporting an index out of range error (there are none in this problem), and giving a list value for x.append(17). Some students wrote 2 rather than 2.0 for (b). I did not take points off for this if you correctly identified the type as float.
2. A company has a policy that its employees are eligible for the deluxe retirement benefits package if they (a) are at least 65 years old; (b) have been on the job at least 40 years; or (c) are at least 60 years old and have been on the job at least 30 years. The criteria are summarized in the decision tree shown in the figure below:

(a) The function below is intended to implement these criteria, taking the employee’s years on the job and age as parameters, and returning True or False according to whether or not they are eligible for the benefits.

```
def eligible(years, age):
    if age >= 65:
        return True
    if years >= 40:
        return True
    if age >= 60:
        if years >= 30:
            return True
        else:
            return False
```

However, the function contains an error. Describe an input on which it gives an incorrect result, and tell how to correct the error.

What happens if an employee is 50 years old and has worked 25 years for the company? This function returns no value, but it is supposed to return False. There are several fixes. The simplest is to add a statement return False as the very last line, indented the same as the three if's.

Some students reported as an error the fact that there is no else following the first and second if's, but this is not a problem, because as soon as a return statement is encountered, we exit the function. Thus
if X
    return Y
return Z

is equivalent to

if X
    return Y
else
    return Z

(b) Alternatively, one can write a version of this function that uses only Boolean operations, and does not use *if* statements at all. The start of this new version is shown below. Describe how to complete it.

```python
def eligible2(years, age):
    sixtyfive = (age >= 65)
    sixty = (age >= 60)
    forty = (years >= 40)
    thirty = (years >= 30)
```

Simplest solution: Add the line

```
return sixtyfive or forty or (sixty and thirty)
```

There are many equivalent correct solutions. You can actually leave off the parentheses, because of precedence rules for boolean operators, but I think they make the formula more readable. Quite a few students wrote something along the lines of

```python
if sixtyfive == True or forty == True or (sixty == True and thirty == True):
    return True
else:
    return False
```

which is correct, and I didn't deduct points for it, but misses much of the point of boolean operations and expressions.
Recall that the function `random.randint(i,j)` returns a random integer uniformly distributed between `i` and `j` inclusive. For example, as we’ve seen before, `random.randint(1,6)` simulates the roll of a single 6-faced die.

3. The following function simulates the toss of a single coin. (You will be asked below about several errors, but there is nothing wrong with the `toss` function!)

```python
import random
def toss():
    if random.randint(0,1)==0:
        return 'heads'
    else:
        return 'tails'
```

The function below is intended to use the `toss` function to simulate the following experiment: Toss a coin repeatedly, and return the number of tosses made until the the coin comes up heads. For example, if the sequence of tosses is

tails, tails, heads

then the function should return 3. The function contains multiple errors (four, by my count). Describe all the errors that you find and the erroneous behavior that they cause, and then correct them.

```python
def toss_until_heads():
    count=0
    result=toss()
    while result != 'heads':
        count+=1
        result=toss()
    return(count)
```

All four errors are probably ones that you yourselves have made. The first is an off-by-one error resulting from an incorrect initial value for `count`: If we get heads on the first toss, then the function should set `count` to 1, but here it is set to 0. Second, the word `heads` should be in quotes, since the function `toss` returns a string ‘heads’ or ‘tails’. Third, we should be calling `toss` repeatedly, so we need a call to it inside the `while` loop. (As written, the function goes into an infinite loop if we get tails on the first toss.) Finally, the function is supposed to return a value, but this function prints a value and returns nothing. A completely correct version is

```python
def toss_until_heads():
    count=1
    result=toss()
    while result != 'heads':
        count+=1
        result=toss()
    return(count)
```
4. Suppose \( v \) is a list of lists of strings, where each entry of \( v \) begins with a student’s name, and is followed by the student’s test scores for a semester, which are also given as strings. For instance \( v \) might look something like this:

\[
[[‘Joe’,’68.2’,’93.1’,’89.5’,’92.7’],
[‘Josephine’,’100.0’,’92.9’,’88.6’,’91.3’],…]
\]

(This is the kind of list that you might get as a result of reading a CSV file into a program.)

(a) The function below takes such a list as a parameter. Describe what it is trying to do. (It contains a bug---see below---but its purpose should be clear enough anyway.) Your description should not repeat the code in English---I mean, do not write ‘it sets a list to the empty list, and then for every i up to one less than the length of v, etc.’ Instead tell me what the end result is: ’It is supposed to return a list that…’. 

```python
def dp(v):
    data=[]
    for i in range(len(v)):
        data.append(v[i][1:])
    w=[v[i][0] for i in range(len(v)) if sum(data[i])/len(data[i])>=90]
    return w
```

The intention is to return a list of the names of all the students whose average test score is at least 90.

(b) When I ran this code I got the following error message.

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

What is the reason for the error, and how should it be fixed? You don’t need to rewrite the entire function, just describe what change needs to made.

The problem is that the numbers in the original list are given as strings, so you need to convert them to floats. This is clear enough, although the exact error message above might be a little confusing: Why does it say that the problem is adding an int and a string? The reason is that if \( x \) is a list, then \( \text{sum}(x) \) is evaluated essentially by executing:
You cannot, in particular, apply `sum` to a list of strings in order to concatenate the strings.

Correcting the error is not completely straightforward. Here is one solution: Replace the line

```python
data.append(v[i][1:])
```

by

```python
data.append([float(v[i][j]) for j in range(1, len(v[i]))])
```

Alternatively, by

```python
row=[]
for j in range(1, len(v[i])):
    row.append(float(v[i][j]))
data.append(row)
```

A very common error was to write something like:

```python
data.append(float(v[i][1:]))
```

but you cannot apply the function `float` to a list in this way.
5. I have a great idea for a new app. I’m going to call it ‘Friendbook’, or ‘Facelist’—actually, I’m still working on the name, but here’s the idea. Each user of my app will have a list of ‘friends’. This relation is symmetric, so if A is on B’s list of friends, then B is on A’s list of friends. The Friendbook world can be represented as a Python dictionary, with each key being the user’s name, and the value associated with the key a list of that user’s friends.

So, some typical key-value pairs in the dictionary might be:

```python
'Snow': ['Doc', 'Grumpy', 'Happy', 'Sleepy', 'Dopey', 'Bashful', 'Sneezy', 'Prince']
'Doc': ['Snow', 'Grumpy', 'Happy', 'Sleepy', 'Dopey', 'Bashful', 'Sneezy']
```

No user can be their own friend, and the list of a user’s friends cannot contain any repeated values.

Now and then, FaceList suggests to users people they might want to become friends with, by proposing someone who is a friend of a friend. Write a function `fof(d, user)` that returns a list of all the friends of friends of user in the dictionary d. (For instance, in the example above, `fof(d, 'Doc')` would return a list that includes 'Prince'). The list that is returned should contain no repeats, nor should it contain the original user or anyone who is already a friend of the user.

In the language of Assignment 7, we are looking for all the users who are at a distance of exactly 2 from the original user. We need a doubly-nested loop, once to loop through the friends of the user, and then to loop through the friends of each friend. We also have to filter out people we encounter whose distance is actually 1 or 0, and also people who have already been added to the list, since we cannot have repetitions. Here is the complete code.

```python
def fof(d, user):
    foflist = []
    for friend in d[user]:
        for ff in d[friend]:
            if ff != user and ff not in d[user] and ff not in foflist:
                foflist.append(ff)
    return foflist
```

Other approaches are possible: You could loop through all the keys in the dictionary and see if user is a friend of a friend of the key you’re looking at, but the above
approach seems more sensible. A common error here was to forget one or more of the properties that had to be filtered out.
6. Pascal’s Triangle is an infinite triangular array of integers whose first few rows look like this:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

The algorithm for calculating the entries of the triangle is the following: We number both the rows and the columns of the triangle starting from 0. Let \( P(i, k) \) denote the entry in row \( i \), column \( k \). (Note that there is no entry if \( i > k \).)

- For every \( k \geq 0 \), \( P(k, 0) = 1 \). (These are the 1’s down the left-hand column.)
- For every \( k \geq 0 \), \( P(k, k) = 1 \). (These are the 1’s down the diagonal in the figure above.)
- All the remaining entries in the table are obtained by adding the entry immediately above, and the entry above and to the left. For example, the 10 indicated in boldface in row 5 is the sum of the entries 6 and 4, also in bold, in the row above. If you inspect the portion of the table in the figure, you will see that every entry is constructed according to this rule.

(a) Write a recursive function \( P(row, column) \) that returns the entry in the given row and column. Your function should contain no looping statement (e.g., for or while) in this function, nor any structure like a list, two-dimensional list, tuple or dictionary. Do not worry about what the function does if \( column > row \) or if either of the parameters is negative.

As noted in class, ‘there is no entry if \( i > k \)’ has the inequality in the wrong direction. This did not seem to create a problem.

The recursive code is a direct translation of the definition:

```python
def P(row, column):
    if column==0 or row==column:
        return 1
    else:
        return P(row-1,column)+P(row-1,column-1)
```
There’s not much leeway here, as the structure of the code is rigidly prescribed in the problem statement. You could break the initial \texttt{if} statement (the base case of the recursion) into two different tests, or leave out the \texttt{else}, but other than that the code will be pretty much the same.

(b) What is the \textbf{total number of calls made to} $P$ as a result of executing the following statement (including the original call)?

\begin{verbatim}
print(P(4,2))
\end{verbatim}

Provide enough detail in your explanation so that I can see how you obtained your answer (Drawing a tree-like diagram will help).

Here is the tree calls to the function. The leaves of the tree are calls that do not generate other calls. There are $11$ calls in all.

The structure of this problem is pretty much identical to the Fibonacci example, and you can see another ‘recursion disaster’ looming, with the same calls being made repeatedly.
7. Note that the function below prints output rather than returning a value.

```python
def what(t, n):
    if len(t) == n:
        print(t)
    else:
        for j in range(n):
            if j not in t:
                what(t + (j,), n)
```

(a) What gets printed when you call

```python
what(((0,1,2),3))
```

```text
(0,1,2)
```

(b) What gets printed when you call

```python
what(((0,1),3))
```

```text
(0,1,2)
```

(c) ...and when you call

```python
what(((0,),3))
```

```text
(0,1,2)
(0,2,1)
```

(d).and when you call

```python
what((,),3)
```

```text
(0,1,2)
(0,2,1)
(1,0,2)
(1,2,0)
(2,0,1)
(2,1,0)
```

(e) Describe in general what

```python
what((,),n)
```

prints.
It prints all the permutations of \((0,1,\ldots,n-1)\).

(f) Now suppose we added the statement

\[ \text{print}(t) \]

immediately after the `else`. What will \(\text{what}((()),3)\) print? (The answer is a bit long, but has a simple pattern.)

\[
()
(0,)
(0,1)
(0,1,2)
(0,2)
(0,2,1)
(1,)
(1,0)
(1,0,2)
(1,2)
(1,2,0)
(2,)
(2,0)
(2,0,1)
(2,1)
(2,1,0)
\]

Most people missed part (c), writing just \((0,1,2)\) for the answer, and it was downhill from there. What was missed is that after the function returns from the recursive call with \(((0,1),3)\) as the parameters, it generates another call with \(((0,2),3)\). I gave partial credit if you pursued the logic correctly after the initial error.

This is just about identical to the eight queens problem; we’re merely dropping the rule that queens can attack diagonally. (So it’s really the ‘eight rooks’ problem.) This simplest version of the backtracking solver prints out a list of all the solutions of the puzzle; the common error was to stop printing when the first solution was found.