Problem:

Ask the user for two integer **n** and **m where m>n**.

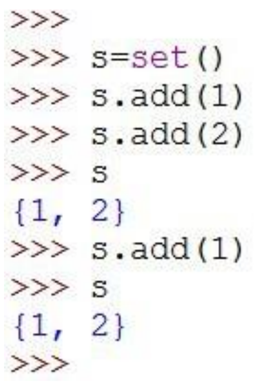
Create a list x of size n. Populate the list with n **random integers** in the range 1through m.

Problem:

Modify the answer to the problem above so that all of the integers in list x are **unique**.

Sets

Like a list, a set is a "container" that gives us a place to store a collection of elements.
Unlike a list, a set can contain ***only one copy*** of any element.



```
>>>
>>> s=set()
>>> s.add(1)
>>> s.add(2)
>>> s
{1, 2}
>>> s.add(1)
>>> s
{1, 2}
>>>
```

We can use Python's built-in ***set*** container to check for duplicates.

We can create a set S with the following syntax:

s=set([iterable])

S=set() creates an initially empty set.

If IT is some iterable (like a set) the s=set(IT) creates a set that is initialized with the elements of IT (for example a list)

The set provides the following operations (among others):

add(elem)
Add element elem to the set.
remove(elem)
Remove element elem from the set. Raises KeyError if elem is not contained in the set.
discard(elem)
Remove element elem from the set if it is present.
pop()
Remove and return an arbitrary element from the set. Raises KeyError if the set is empty.
clear()
Remove all elements from the set.
len(s)
Return the cardinality (number of elements) of set s.
x in s
Test x for membership in s.
x not in s
Test x for non-membership in s.

Problem:

Ask the user for three integer **n** and **m and k, where k>m*n**.

Create a two dimensional table (list of lists) x of size n*m. Populate the list with n*m **unique random integers** in the range 1through k.

Problem:

Write a function get_row(x,i) which **takes a square matrix** (=table=list of lists) and **returns a list** containing the ith row of matrix X.

Problem:

Write a function get_col(x,i) which **takes a square matrix** (=table=list of lists) and **returns a list** containing the ith column of matrix X.

Problem:

Write a function to calculate and return the sum of all the elements on the “main diagonal” of the square two dimensional list x passed to it as an argument. This is the diagonal going from the upper left to the bottom right.

Problem:

Like the problem above, calculate and return the diagonal sum, but for the elements along the diagonal going from the top right to the lower left.

List Comprehensions

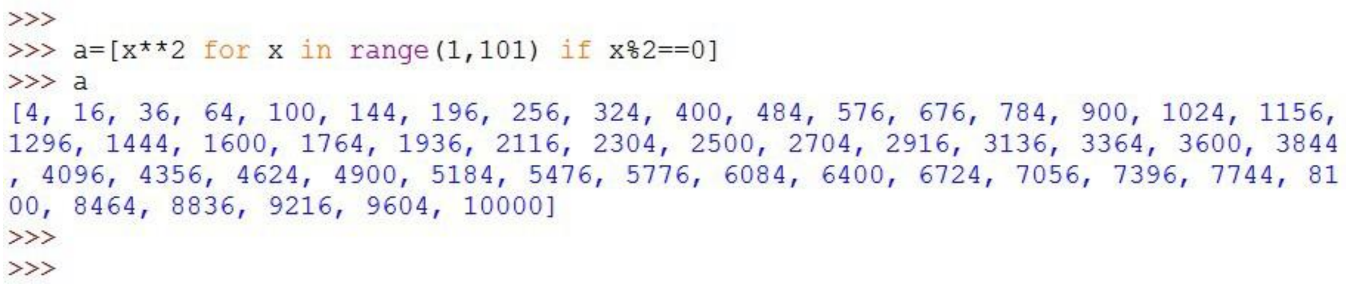
A list comprehension is a very compact and useful notation that Python provides for **creating lists**.

It is very similar to the notation that is uses in math for specifying sets. For example it is pretty clear what this means:

$$\{x^2 : 1 \leq x \leq 100 \text{ , if x is even}\}$$

It is the set of the squares of the even numbers between 1 and 100.

Here is a list comprehension that creates a list with the same numbers:



```
>>>
>>> a=[x**2 for x in range(1,101) if x%2==0]
>>> a
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784, 900, 1024, 1156,
1296, 1444, 1600, 1764, 1936, 2116, 2304, 2500, 2704, 2916, 3136, 3364, 3600, 3844
, 4096, 4356, 4624, 4900, 5184, 5476, 5776, 6084, 6400, 6724, 7056, 7396, 7744, 81
00, 8464, 8836, 9216, 9604, 10000]
>>>
>>>
```

Notice that this is equivalent to the following:

```
a=[]
for x in range(1,101):
    if x%2==0:
        a.append(x)
```

The list comprehension notation is more compact and much clearer (once you get used to it) than the equivalent code above

The general form has three components:

[expression using a value from an iterable   for iterable   if conditions]

(1)   (2)   (3)

Some more examples:

Example:

Assume that we have a function prime(i) which returns True if i is s prime number and False otherwise.

Create a list of all the primes between 2 and 120.

y = [k   for k in range(2, 121)   if prime(k)   ]

Example:

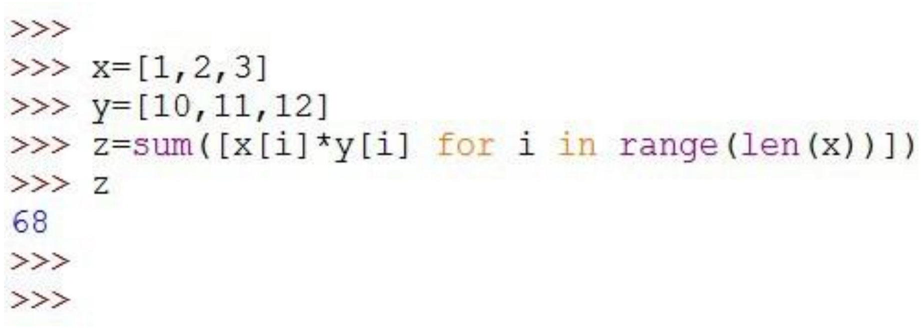
Let x and y be two lists of numbers, both of length n. To get the **dot product** of x and y we do the following:

- form the pairs x[i]*y[i] for 0 ≤ i < n
- sum up all the pairs.

For example:

x=[1,2,3] and y=[10,11,12] then x·y=1*10+2*11+3*12.

We can write this as list comprehension like this:



```
>>>
>>> x=[1,2,3]
>>> y=[10,11,12]
>>> z=sum([x[i]*y[i] for i in range(len(x))])
>>> z
68
>>>
>>>
```

Example:

Let z be a list of 20 integers, and we want a **list of tuples** giving the value and position of each even number in the list. The following will do this:

```
>>> z
[13, 14, 7, 12, 11, 3, 18, 20, 5, 7, 19, 17, 10, 16, 14, 12, 5, 19, 15, 1]
>>> k=[ ( i , z[i] )  for i in range(20)    if z[i]%2==0 ]
>>> k
[(1, 14), (3, 12), (6, 18), (7, 20), (12, 10), (13, 16), (14, 14), (15, 12)]
>>>
```

Example:

Write a function get_row(x,i) which **takes a square matrix** (=table=list of lists) and **returns a list** containing the element in the ith **row** of matrix X.

Using a list comprehension we could write:

```
def get_row(x,i):
    return [ x[i][j] for j in range(len(x)) ]
```

Or …we could also write

```
def get_row(x,i):
    return x[i]
```

since the ith row of x is just the ith sublist of x.

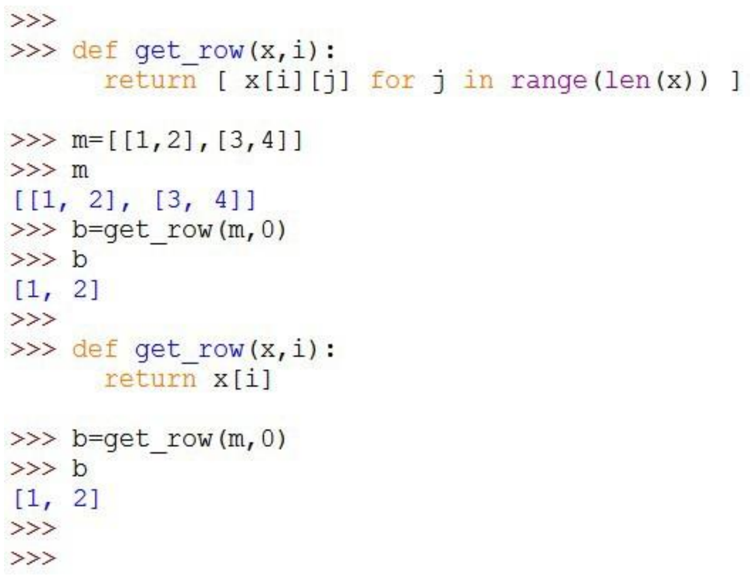
However there is an **important difference between these two implementations** of get_row().

The first version creates a new list made up of the elements of row i of matrix X.

The second one, which is much faster, just returns a reference to the ith row of matrix X.

In the following example, we call **both** versions of get_row(m,0) on the matrix **m=[[1,2],[3,4]].**

Notice that both versions return what **seems to be** the same result: [1, 2]



```
>>>
>>> def get_row(x,i):
        return [ x[i][j] for j in range(len(x)) ]

>>> m=[[1,2],[3,4]]
>>> m
[[1, 2], [3, 4]]
>>> b=get_row(m,0)
>>> b
[1, 2]
>>>
>>> def get_row(x,i):
        return x[i]

>>> b=get_row(m,0)
>>> b
[1, 2]
>>>
>>>
```

Question: What is the important **practical difference** between the two versions?

Hint: What happens when we change list b after it is returned from get_row(). Why does this happen?

Example:

Write a function get_col(x,i) which **takes a square matrix** (=table=list of lists) and **returns a list** containing the items in the ith **column** of matrix X.

```
def get_col(x,i):
    return [ x[j][i] for j in range(len(x)) ]
```

Problem:

Write a function sum_col(x,i) which **takes a square matrix** (=table=list of lists) and **the sum of** the items in the ith **column** of matrix X. Use a list comprehension.

Problem:

Write a function diag_diff(x,i) which **takes a square matrix** (=table=list of lists) and **the difference of two main diagonals of matrix X.** In other words, let d1 be the diagonal of X from upper left to bottom right, and d2 to be the diagonal from upper right to lower left. The function returns d1-d2. Use list comprehensions**.**

Problem:

Early versions of Python distinguished between int and long. ints were represented natively on the hardware and longs were lists of integers. From version 3.0 and on all integer types are represented as longs.

Write a function **make_int (x: str) -> list:**

that takes a string of digits and returns a list of those digits. Use a list comprehension.

Problem:

Write a function int_add(x,y) that takes two “integers” as represented above, and returns their sum.

Note that these “integers” needn’t be the same size. You can simplify your program by padding the shorter one with zeros (on the left).

Note as well, there might be a final carry that will increase the answer by one digit more than the addends.

Problem:

Write a function matrix_add(x,y) that takes two square “matrices” and returns their sum.

Problem:

Write a function matrix_mult(x,y) that takes two square “matrices” and returns their product.

More list Comprehensions – “two-dimensional” lists

In Python, you can use **list comprehensions** to create 2-dimensional lists (i.e., lists of lists) in a concise and efficient way.

Basic Syntax for 2D List Comprehensions

The general syntax for creating a 2D list using list comprehension is:

[[expression for column in range(num_columns)] for row in range(num_rows)]

Here:

- The outer list comprehension creates each **row**.
- The inner list comprehension creates each **element in the row** (i.e., each **column**).

Problem 1: Create a 2D List of Zeros (3x3 Matrix)

Create a 3x3 matrix (3 rows and 3 columns) filled with `0`s using list comprehension.

```
[[0, 0, 0],
[0, 0, 0],
[0, 0, 0]]
```

Problem 2: Create a 2D List with Incremental Values

```
[[0, 1, 2],
 [3, 4, 5],
 [6, 7, 8]]
```

Problem 3: Craete an Identity Matrix (5x5 Matrix)

An **identity matrix** is a square matrix where all the diagonal elements are `1` and all off-diagonal elements are `0`.

```
[[1, 0, 0, 0, 0],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 0, 1]]
```

Problem 4: Generate a 2D list where each element is the product of its row and column indices:

```
[[0, 0, 0, 0],
 [0, 1, 2, 3],
 [0, 2, 4, 6],
 [0, 3, 6, 9]]
```

Solutions

Problem 1.

```
matrix = [[0 for col in range(3)] for row in range(3)]
print(matrix)
```

```
[[0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]]
```

Problem 2.

```
matrix = [[col + row * 3 for col in range(3)] for row in range(3)]
print(matrix)
```

```
[[0, 1, 2],
 [3, 4, 5],
 [6, 7, 8]]
```

Problem 3.

```
identity_matrix = [[1 if row == col else 0 for col in range(5)] for row in range(5)]
print(identity_matrix)[[1, 0, 0, 0, 0],
```

```
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

Problem 4.

```
matrix = [[row * col for col in range(4)] for row in range(4)]
print(matrix)
```

```
[[0, 0, 0, 0],
 [0, 1, 2, 3],
 [0, 2, 4, 6],
 [0, 3, 6, 9]]
```

Problem:

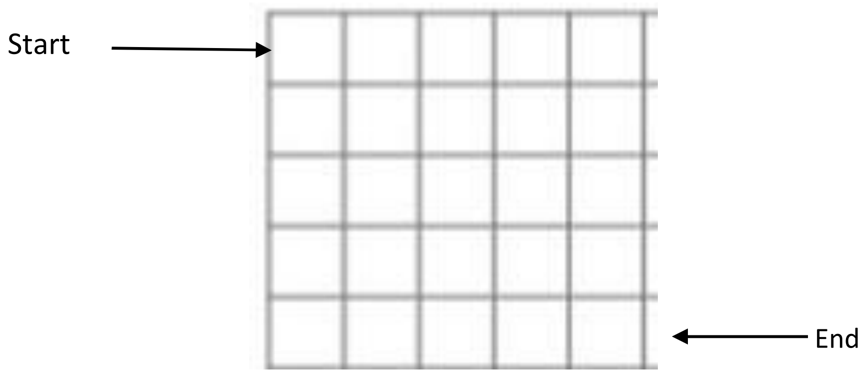
Now, here is a definition: A **saddle point** in a 2 dimensional square table is an entry in the table whose value is the minimum in its row **and** maximum in its column. In the table below, 0 is a saddle point.

2	0
0	-2

Write a function to find a saddle point in a 2 dimensional table in any two dimensional square of integers, if one exists. If a saddle point was found return a triple (value, x pos, y pos). If a saddle point was not found return “not found”

Problem:

. Consider the 5X5 square below:



We would like to have a robot travel from the start square at the upper left to the end square at the lower right. At each square the robot has only one of two moves:

- It may go one square to the right or
- It may go one square down.

Write a program that determines in how many ways may this be done? In other words, how many paths are there from start to end with each move restricted as above?

Solution:

```
#robot paths
a=[5*[0] for i in range (5)]
#set up the left and top boundaries of the table (5X5 square array)
for i in range(5):
    a[i][0]=1
    a[0][i]=1

for i in range(1,5):
    for j in range(1,5):
        a[i][j]=a[i-1][j]+a[i][j-1]

print(a[4][4])
```

Here is another approach. Can you figure out how this works? What happened to the table??

```
n=int(input("Please enter the 'size' of the array, n= "))

def num_paths(n,i,j):
    if i==0 or j ==0:
        return 1
    return num_paths(n,i-1,j)+num_paths(n,i,j-1)

print(num_paths(n, n-1, n-1))
```

Problem:

Write a program to generate all the eight-digit base 8 integers from 00000000➔77777777

What is a base eight integer?

The following code will do it.

```
for i0 in range(8):
    for i1 in range(8):
        for i2 in range(8):
            for i3 in range(8):
                for i4 in range(8):
                    for i5 in range(8):
                        for i6 in range(8):
                            for i7 in range(8):
                                q=[i0,i1,i2,i3,i4,i5,i6,i7]
                                print(q)
```

Problem:

Write a function get_num(n) which returns a list of length 8 representing the base eight representation of the decimal number n.

Problem:

Remember the Eight Queens Problem? Using the code above, we can generate all 8^8 configurations and print only those that have neither a row nor a diagonal conflict.

The algorithm

The row test (its trivial in Python)

The diagonal test

Problem:

What does the following code do? (and how?)

```
from itertools import permutations

col=[0,1,2,3,4,5,6,7]

for vec in permutations(col):
    if (8==len(set(vec[i]+i for i in col))
        == len(set(vec[i]-i for i in col))):
         print(vec)
```

Here is another way:

```
def diagonal_threat(q):



from itertools import permutations
candidates=permutations([0,1,2,3,4,5,6,7])

count=0
for p in candidates:
    if diagonal_threat(p):
      continue
    count+=1
    print('Solution number ', count,': ',p)
    print()
```

Before we move on to strings, let's revisit

Unpacking Sequences and Iterables

We covered some of this before, but we review here and look at some additional uses.

Unpacking is important because it avoids unnecessary and error-prone use of indexes to extract elements from sequences. Also, unpacking works with any iterable object as the data source—including iterators, which don't support index notation ([]). The only requirement is that the iterable yields exactly one item per variable in the receiving end, unless you use a star (*) to capture excess items, as explained below

Parallel Assignment

The most visible form of unpacking is parallel assignment; that is, assigning items from an iterable to a tuple of variables, as you can see in this example:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates  # unpacking
>>> latitude
33.9425
>>> longitude
-118.408056
```

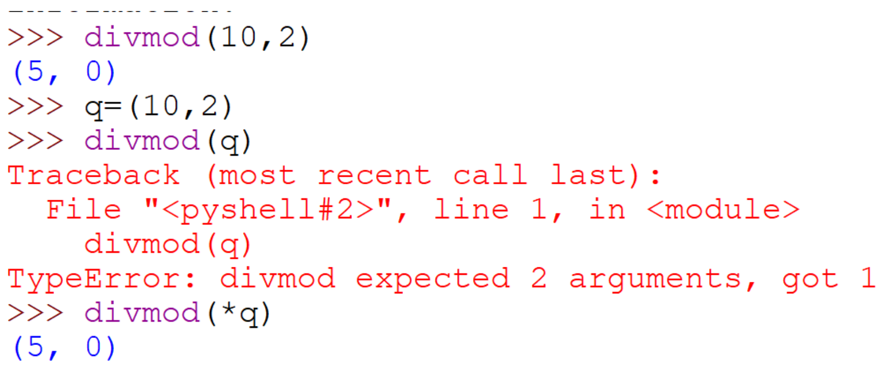
An elegant application of unpacking is swapping the values of variables without using a temporary variable:

```
>>> b, a = a, b
```

Another example of unpacking is **prefixing an argument with * when calling a function:**

```
>>> divmod(20, 8) # returns the divisor and remainder
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t) # in this case the t variable is the "args"
>>> quotient, remainder
(2, 4)
```

In other wors:



```
>>> divmod(10,2)
(5, 0)
>>> q=(10,2)
>>> divmod(q)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    divmod(q)
TypeError: divmod expected 2 arguments, got 1
>>> divmod(*q)
(5, 0)
```

The preceding code shows another use of unpacking: allowing functions to return multiple values in a way that is convenient to the caller.

As another example**, the os.path.split() function builds a tuple (path, last_part) from a filesystem path:**

The `os.path.split()` function in Python is used to split a filesystem path into a tuple consisting of two parts:

1. Everything except the last component of the path

2. The last component of the path

This can be useful for extracting the file name from a full file path, or for separating a directory path from a filename.

The `os.path.split()` function is helpful for manipulating and extracting specific parts of filesystem paths, making it easier to work with files and directories in Python programs.

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/id_rsa.pub') #note the "_" at the beginning.
>>> filename # matches the last part
'id_rsa.pub'
```

Another way of using just some of the items when unpacking is to use the * syntax, as we'll see right away.

Using * to Grab Excess Items

Defining function parameters with *args to grab arbitrary excess arguments is a classic Python feature.

We worked on the following example where we passed in an arbitrary number of arguments:

Problem: Write a function "addem" that returns the sum of an arbitrary number of numbers.

so addem(1,2) ==>3 and addem(1,2,3)==> 6.

In Python 3, this idea was **extended to apply to parallel assignment** as well:

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

Note that the excess items get put into a list.

In the context of parallel assignment, the * prefix can be applied to exactly one variable, but it can appear in any position:

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

Unpacking with * in **Function Calls and Sequence Literals**

In function calls, we can use * multiple times:

```
>>> def fun(a, b, c, d, *rest):
...     return a, b, c, d, rest
...
>>> fun(*[1, 2], 3, *range(4, 7))
(1, 2, 3, 4, (5, 6))
```

Notice the 5 and 6 are put into a tuple because of the * in the function header**.**

The * can also be used when defining list, tuple, or set literals, as shown in the following examples.

```
>>> *range(4), 4
(0, 1, 2, 3, 4) # all got unpacked into a tuple by default

>>> [*range(4), 4]
[0, 1, 2, 3, 4] # unpacked into a list because …
>>> {*range(4), 4, *(5, 6, 7)}
{0, 1, 2, 3, 4, 5, 6, 7} # and to a set because …
```

Finally, a powerful feature of tuple unpacking is that it works with nested structures.

Nested Unpacking

The target (what you are unpacking into) of an unpacking can use nesting, e.g., (a, b, (c, d)). Python will do the right thing if the value has the same nesting structure.

Example: Unpacking nested tuples to access the longitude

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

def main():
    print(f'{"":15} | {"latitude":>9} | {"longitude":>9}')
    for name, _, _, (lat, lon) in metro_areas:
        if lon <= 0:
            print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
```

Each tuple holds a record with four fields, the last of which is a coordinate pair.

By assigning the last field to a nested tuple, we unpack the coordinates.

The lon <= 0: test selects only cities in the Western hemisphere.

The output is:

```
                |  latitude | longitude
Mexico City     |   19.4333 |  -99.1333
New York-Newark |   40.8086 |  -74.0204
São Paulo       |  -23.5478 |  -46.6358
```

Another use of * is a using zip(*B) to create a transpose of a matrix or any iterable B

First let’s look at zip.

The `zip` function in Python takes two or more sequences (like lists or tuples) and "zips" them together into a single iterable of **tuples**, where each tuple contains the elements from the input iterables that are in the same positions.

Basic Usage
If you provide two lists of equal length to `zip`, it pairs up the elements at each position:

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
zipped = zip(a, b)
print(list(zipped))  # Output: [(1, 'a'), (2, 'b'), (3, 'c')]
```

Handling Different Lengths
If the input iterables are not of the same length, `zip` stops creating tuples when the shortest input iterable is exhausted:

```
a = [1, 2, 3, 4]
b = ['a', 'b']
zipped = zip(a, b)
print(list(zipped))  # Output: [(1, 'a'), (2, 'b')]
```

Unzipping

You can also use `zip` to "unzip" a list of tuples into separate lists:

```
pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
numbers, letters = zip(*pairs)
print(numbers)  # Output: (1, 2, 3)
print(letters)  # Output: ('a', 'b', 'c')
```

In the above example, the `*` operator is used to unpack the list of tuples, and `zip` then groups all the first elements together, all the second elements together, etc.

Here's a breakdown of how it works:

1. Initialization of the `pairs` List
   pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
   This line creates a list of tuples, where each tuple contains a number and a corresponding letter.

2. Using `zip(*pairs)`:

~~~
   numbers, letters = zip(*pairs)
   ```
~~~

   `*pairs` is the unpacking operator, which unpacks the list `pairs` into its individual elements (tuples). So, after unpacking, it's as if you're passing each tuple as a separate argument to the `zip` function: `zip((1, 'a'), (2, 'b'), (3, 'c'))`.

The `zip` function then combines the elements of these tuples based on their index. It takes all the first elements of these tuples and groups them together, and does the same with the second elements. So, you get two groups: one for numbers `(1, 2, 3)` and one for letters `('a', 'b', 'c')`.

3.The output of `zip` is a zip object that yields tuples. The first tuple contains all the first elements of the original tuples (`(1, 2, 3)`), and the second tuple contains all the second elements (`('a', 'b', 'c')`).
These tuples are then unpacked into two variables, `numbers` and `letters`.

4. Printing the Results:
`print(numbers)` outputs `(1, 2, 3)` which are the grouped first elements of each tuple in `pairs`.
`print(letters)` outputs `('a', 'b', 'c')` which are the grouped second elements of each tuple in `pairs`.

This technique is often used for transposing data structures or for parallel iteration in Python.

Using `zip` in a Loop

You can use `zip` in a for loop to iterate over multiple lists in parallel:

```
for number, letter in zip([1, 2, 3], ['a', 'b', 'c']):
    print(number, letter)
```

This would output:

```
1 a
2 b
3 c
```

Creating Dictionaries

You can also use `zip` to create dictionaries when you have two lists: one for keys and one for values:

```
keys = ['a', 'b', 'c']
values = [1, 2, 3]
dictionary = dict(zip(keys, values))
print(dictionary)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

So, zip can be used for a variety of purposes, including **transposing matrices**, iterating in parallel, and creating dictionaries, among others.

Transposing matrices

How?

The `zip(*B)` expression in Python is used for unpacking and transposing the elements of an iterable `B`. The transpose of a matrix is obtained by swapping the rows and columns. For the matrix `B` you provided, the transpose would be achieved by turning the rows of `B` into columns.

Here is the transpose of matrix `B`:

```
B = [
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
]
```

```
Transpose of B = [
   [1, 4, 7],
   [2, 5, 8],
   [3, 6, 9]
]
```

In the transposed matrix, the first row `[1, 2, 3]` of `B` becomes the first column `[1, 4, 7]` of the transposed matrix, the second row `[4, 5, 6]` becomes the second column `[2, 5, 8]`, and the third row `[7, 8, 9]` becomes the third column `[3, 6, 9]`.

So, if

```
B = [
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
]
```

When you call `zip(*B)`, it is equivalent to calling `zip([1, 2, 3], [4, 5, 6], [7, 8, 9])`.

The `zip` function will group the elements with the same index from each sublist together, effectively transposing the rows and columns.

For the above example, after transposing, it would become:

```
 [
   (1, 4, 7),
   (2, 5, 8),
   (3, 6, 9)
]
```

Aside: Remember, when you zip you get tuples. You can convert the tuples back to lists (if needed) by using a list comprehension:

```
transposed = [list(group) for group in zip(*B)]
print(transposed)
# Output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

This is a concise way to transpose a matrix in Python, making use of the built-in `zip` function.

Here is an interesting use of this.

```
def matrix_mult(A,B):
    """
    Assumed that count(cols) == count(rows).
    Use zip(*B) to find transpose matrix of B.
    """
    dot = lambda r,c: sum([r[i] * c[i] for i in range(len(r))])

    return [[dot(ri,cj) for cj in zip(*B)] for ri in A]
```

A very “pythonic” matrix multiplier.

matrix_mult(A, B) is designed to perform matrix multiplication on two matrices `A` and `B`. Let's break down how it works:

```
def matrix_mult(A, B):
```

This defines a function named `matrix_mult` that takes two arguments, `A` and `B`, which are expected to be matrices represented as lists of lists in Python.

We assumed that count(cols) == count(rows) and we use zip(*B) to find transpose matrix of B.

The function uses `zip(*B)` to obtain the transpose of matrix `B`. In matrix multiplication, you multiply rows of the first matrix with columns of the second matrix. Transposing `B` makes it easier to access its columns as rows.

The `dot` Lambda Function:

```
dot = lambda r,c: sum([r[i] * c[i] for i in range(len(r))])
```

This lambda function, named `dot`, calculates the dot product of two vectors. It takes two arguments `r` and `c`, which represent a row from `A` and a column from `B` (or a row from the transpose of `B`). The dot product is calculated by multiplying corresponding elements and then summing up these products.

Matrix Multiplication Logic:

return [[dot(ri,cj) for cj in zip(*B)] for ri in A]

This is a nested list comprehension which forms the core of the matrix multiplication:

- `for ri in A`: This outer loop iterates over each row `ri` in matrix `A`.
- `for cj in zip(*B)`: This inner loop iterates over each transposed row (which is effectively a column) `cj` in matrix `B`.
- `dot(ri, cj)`: For each pair of row `ri` from `A` and column `cj` from `B`, the dot product is calculated.

The outer list comprehension collects these dot products into rows, and the inner list comprehension assembles the rows into the resulting matrix.

The function returns a new matrix which is the product of matrix `A` and matrix `B`.

So, for example:
If `A` is a 3x2 matrix and `B` is a 2x3 matrix, `matrix_mult(A, B)` will return a 3x3 matrix where each element `(i, j)` is the dot product of the `i`-th row of `A` and the `j`-th column of `B`.

More on Strings … and Files

We have been using strings all along. Some of the things we have seen include:

- Strings are immutable.
- Strings are iterables so we can use "for".
- Since strings are sequences, we can access elements and substrings them [].
- We can concatenate strings: s1+s2.

But Python provides many many functions for working with strings. Let's check out some of the functions as described in the online documentation. The full list is in the Python Library Reference documentation in section 4.6.1.

Here are some of the most useful with definitions and examples from the Library Reference. .

`str.count`(*sub*[, *start*[, *end*]])

> Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

`str.find`(*sub*[, *start*[, *end*]])

> Return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

`str.join`(*iterable*)

> **Return a string which is the concatenation of the strings in the *iterable*** *iterable*. A `TypeError` will be raised if there are any non-string values in *seq*, including `bytes` objects. The separator between elements is the string providing this method.

`str.replace`(*old*, *new*[, *count*])

> Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.split`([*sep*[, *maxsplit*]])

> **Return a list of the words in the string**, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified, then there is no limit on the number of splits (all possible splits are made).
>
> If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example, `' 1  2   3  '.split()` returns `['1', '2', '3']`, and `'  1  2   3  '.split(None, 1)` returns `['1', '2   3`

`str.splitlines`([*keepends*])

Return a list of the lines in the string, breaking at line boundaries. **Line breaks are not included in the resulting list** unless *keepends* is given and true.

For example:
text = "Hello\nWorld!\rThis is an example.\r\nEnjoy!"
lines = text.splitlines()
print(lines)
# Output: ['Hello', 'World!', 'This is an example.', 'Enjoy!']

text = "Hello\nWorld!\rThis is an example.\r\nEnjoy!"
lines = text.splitlines(True)
print(lines)
# Output: ['Hello\n', 'World!\r', 'This is an example.\r\n', 'Enjoy!']

Line boundaries include:

'\n': Line Feed
'\r': Carriage Return
'\r\n': Carriage Return + Line Feed

`str.rstrip`([*chars*])

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. **The *chars* argument is not a suffix; rather, all combinations of its values are stripped:**

There are also functions lstrip() and rstrip() that act in the expected way.

`str.upper`()

Return a copy of the string converted to uppercase.

`str.lower`()

Return a copy of the string converted to lowercase

Problem:

Write a function that reverses a string. Remember, a string is not mutable! Do it two ways: using slicing, and using a loop.

Answer:

Another way:

The **`reversed()`** function is a built-in Python function used to reverse the elements of a sequence, such as a list, tuple, or string. When applied to a string, it returns an **iterator** that produces the characters of the string in reverse order.

result = reversed("Hello, World!")

For example, `reversed("Hello, World!")` returns an iterator that will produce the characters of the string `"Hello, World!"` in reverse order when iterated over.

So,

```
def reverse_string(s):
    return ''.join(reversed(s))
```

- `reversed(s)` creates a reverse iterator of the string `s`.
- `''.join(reversed(s))` takes this iterator and joins its items (characters of the string `s`) together into a new string, effectively reversing the string.

The `reversed()` function is useful when you want to iterate over the items of a sequence in reverse order without actually creating a reversed copy of the sequence.

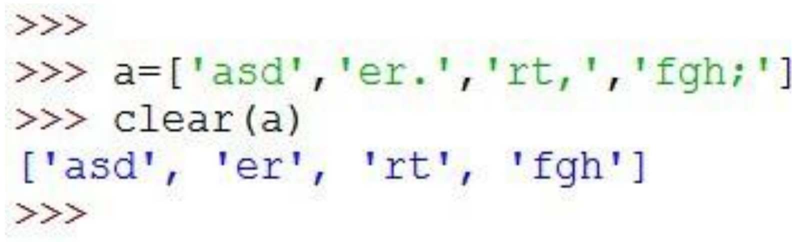
Problem:

Write a function one_upper(s) which returns True if **exactly one** character in string s is capitalized, and False otherwise. You can assume that the string s contains only alphabetic characters and no blanks. You might want to consider other string functions from the documentation.

Answer:

Problem:

Write a function clear(x) where x is a list of “words”. Each word is a string that might have either a ‘.’ ‘,’, or’;’ tacked on at the end. clear() will return a list with the original words stripped of the punctuation.



```
>>>
>>> a=['asd','er.','rt,','fgh;']
>>> clear(a)
['asd', 'er', 'rt', 'fgh']
>>>
```

Answer:

```
def clear(x):
    cleaned_words = []
    for word in x:
        if word[-1] in ('.', ',', ';'):
            cleaned_word = word[:-1]
        else:
            cleaned_word = word
        cleaned_words.append(cleaned_word)
    return cleaned_words
```

and using a list comprehension we get a shorter …

```
def clear(x):
    return [word.rstrip('.,;') for word in x]
```

Actually, the list comprehension using the member function is much nicer and clearer.

Question: Do the two versions of clear behave exactly the same?

Question: What about this version**?**

```
def clear(x):
    return [word[:-1] if word[-1] in '.,;' else word for word in x]
```

Problem:

What about removing the unwanted characters from the left end? What about from both ends? What about from anywhere in the string?

Answer:

```
def clear_left(x):
   return [word.lstrip('.,;') for word in x]

 def clear_both_ends(x):
    return [word.strip('.,;') for word in x]

def clear_anywhere(x):
   return [''.join(char  for char in word  if char not in '.,;')  for word in x]
```

Problem:

Problem:

Write a **function scrape(s)** which take a string s representing the HTML of a web page and returns a list of all links found on the page. Explain in detail how this works.

Answer:

Problem: Count the ones.

Write a program that prompts the user for some integer n, and calculates the number of occurrences of the digit ‘1’ in all the numbers from 1-n inclusive.

For example, if n=20 the program will print 12 since

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20.

Files: Input and Output

Basic file operations

output = open('C:\spam', 'w')	Create output file ('w'means write)
input = open('data', 'r')	Create input file ('r'means read)
input = open('data')	Same as prior line ('r'is the default)
aString = input.read()	Read entire file into a single string
aString = input.read(N)	Read up to next Ncharacters (or bytes) into a string
aString = input.readline()	Read next line (including \nnewline) into a string
aList = input.readlines()	Read entire file into list of line strings (with \n)
output.write(aString)	Write a string of characters (or bytes) into file
output.writelines(aList)	Write all line strings in a list into file
print(value, …file='filename')	**Write to file "filename" instead of to the screen**
output.close()	Manual close (done for you when file is collected)
output.flush()	Flush output buffer to disk without closing any File.
seek(N)	Change file position to offset Nfor next operation
for line in open('data'):*use line*	File iterators read line by line
open('f.txt', encoding='latin-1')	Python 3.0 Unicode text files (strstrings)
open('f.bin', 'rb')	Python 3.0 binary bytes files (bytesstrings)

Reading from files.

Reading a file line by line using a "context manager" - <u>with</u> more examples later.

```
with open('data.txt') as file:
    for line in file:
        print(line, end='') # end='' omits the extra newline
```

- The open() function returns a new file object.
- The with statement that precedes it declares a block of statements (or context) where the file is going to be used.

Once control leaves this block, the file is automatically closed. If you don't use the with statement, the code would need to look like this:

```
file = open('data.txt')
for line in file:
    print(line, end='') # end='' omits the extra newline
file.close()
```

It's easy to forget the extra step of calling close() so it's better to use the with statement and have the file closed for you.

The for loop iterates line-by-line over the file until no more data is available.

If you want to read the file in its entirety as a string, use the read() method like this:

```
with open('data.txt') as file:
    data = file.read()
```

If you want to read a large file in chunks, give a size hint to the read() method as follows:

```
with open('data.txt') as file:
    while (chunk := file.read(10000)):
        print(chunk, end='')
```

The **:=** operator (the walrus operator – we saw this earlier) assigns to a variable and returns its value so that it can be tested by the while loop to break out. When the end of a file is reached, read() returns an empty string. We have this in C++ e.g. **while(cin>>a) {…}**

An alternate way to write the above function is using break:

```
with open('data.txt') as file:
    while True:
        chunk = file.read(10000)
        if not chunk:
            break
        print(chunk, end='')
```

Reading data typed interactively in the console.

To do that, use the input() function.

For example:

```
name = input('Enter your name : ')
print('Hello', name)
```

The input() function returns all of the typed text up to the terminating newline, which is not included.

Writing to files.

To make the output of a program go to a file, supply a file argument to the print() function:

```
with open('out.txt', 'wt') as out:
    while year <= numyears:
        principal = principal * (1 + rate)
        print(f'{year:>3d} {principal:0.2f}', file=out)
        year += 1
```

In addition, file objects support a write() method that can be used to write string data.

For example, the print() function in the previous example could have been written this way:

```
out.write(f'{year:3d} {principal:0.2f}\n')
```

Problem:

Create this file (bears.txt) in your python directory:

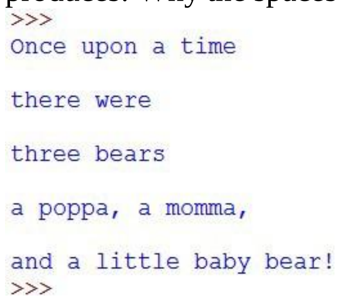
```
Once upon a time
there were
three bears,
a poppa, a momma,
and a little baby bear!
```

Read the file line by line and print it out.

Answer:

```
f=open('bears.txt')
for i in f:
     print(i)
```

produces: Why the spaces between the lines?



```
>>>
Once upon a time

there were

three bears

a poppa, a momma,

and a little baby bear!
>>>
```

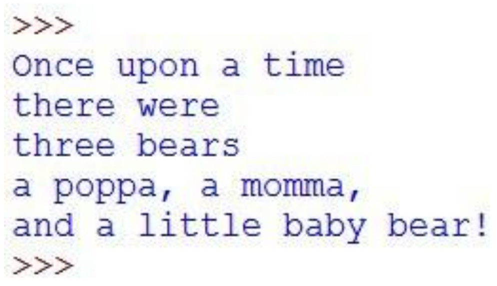
Problem:

Read the file into one string and print the string.

Answer:

```
f=open('bears.txt')
z=f.read()
print(z)
```

produces: What happened to the blank lines?



```
>>>
Once upon a time
there were
three bears
a poppa, a momma,
and a little baby bear!
>>>
```

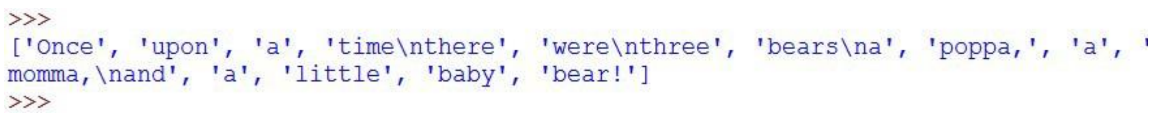
Problem:

Read the file into a string s, separate the words in to a list (use ‘ ‘ to indicate the separator between words. Print out the list.

Answer:

```
f=open('bears.txt')
z=f.read()
z=z.split(' ')
print(z)
```

produces:



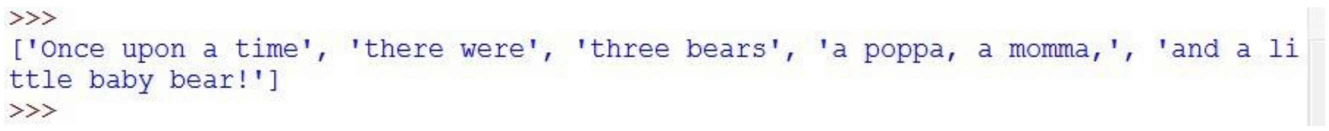
```
>>>
['Once', 'upon', 'a', 'time\nthere', 'were\nthree', 'bears\na', 'poppa,', 'a', '
momma,\nand', 'a', 'little', 'baby', 'bear!']
>>>
```

Note the ‘\n’ , the newline character. This causes a like break.

but …

```
f=open('bears.txt')
z=f.read()
z=z.splitlines()
print(z)
```

produces:



```
>>>
['Once upon a time', 'there were', 'three bears', 'a poppa, a momma,', 'and a li
ttle baby bear!']
>>>
```

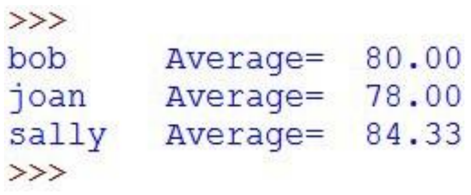
Notice that the newline characters are gone.

Problem:

Create a text file, 'grades.txt', with the following data.

```
Bob 78 98 67 77
Joan 78
Sally 90 97 77 56 88 98
```

Write a program to read this file and print out the students name followed by their average on all exams.



```
>>>
bob     Average=  80.00
joan    Average=  78.00
sally   Average=  84.33
>>>
```

Answer: How about this? …

```
#Student averages
f=open('grades.txt')
for i in f:
  s=i.split()
  average=sum([int(s[i]) for i in range(1,len(s))])/(len(s)-1)
  print(format(s[0],'<7s'),'Average= ',format(average,'.2f'))
```

or this:

```
def calculate_average(scores):
    return sum(scores) / len(scores)

with open("student_scores.txt", mode="r") as file:
    # Iterate through lines in the file
    for line in file:
        # Split the line into words
        words = line.split()

        # Extract the student's name and exam scores
        name = words[0]
        scores = [int(score) for score in words[1:]]

        # Calculate the average score
        average_score = calculate_average(scores)

        # Print the student's name and their average score
        print(f"Student: {name}, Average Score: {average_score:.2f}")
```

Problem:

Modify the above program so that it writes the result to a file called averages.

Answer:

Problem:

In the following example bnames.txt is a file with students first name followed by last name followed by midterm score and final score and writes to an output file the first and last names, followed by the midterm and final grades, calculates the average and the student's deviation from the class average.

Notice that the output file is sorted by last name and then by first within last.

Input file:

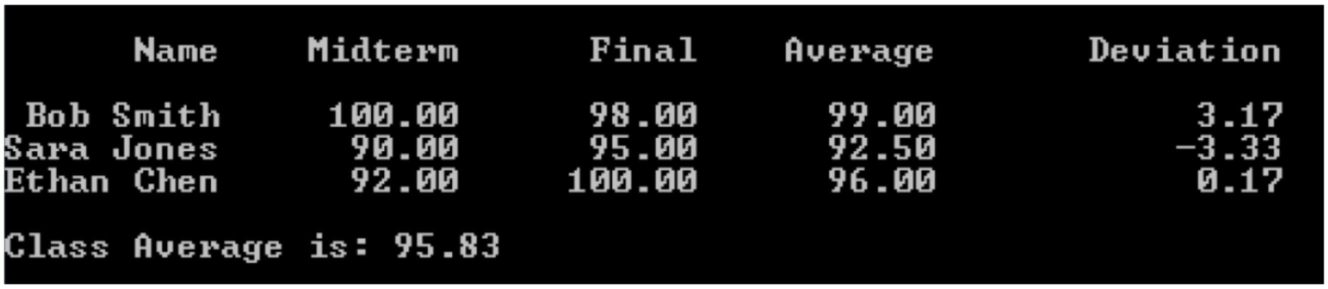
```
bob smith 45 89
sally jones 89 76
alan smith 45 89
```

yields

Output file:

```
sally    jones    89.0  76.0  82.5  10.333333333333329
alan     smith    45.0  89.0  67.0  -5.166666666666671
bob      smith    45.0  89.0  67.0  -5.166666666666671
```

As before your program will then **read** results.txt and produce a report like this:



```
      Name     Midterm       Final     Average          Deviation

 Bob Smith      100.00       98.00       99.00               3.17
Sara Jones       90.00       95.00       92.50              -3.33
Ethan Chen       92.00      100.00       96.00               0.17

Class Average is: 95.83
```

But notice the output is not in the order expected (and the names are different ..). How do we get the correct order?

```
cav=0  # class average

def last_then_first(x):
    return x[1]+x[0]

grades=[]  # This will be a 2 dimensional table with the grades of all the
students

for line in open('bnames.txt'):  # The file with the student grades, one/line
    line=line.split()
    line[2]=float(line[2])
    line[3]=float(line[3])
    grades.append(line)
    cav+=(line[2]+line[3])/2

cav=cav/len(grades)

outfile=open('scores.txt','w')
grades.sort(key=last_then_first)
for line in grades:
    av=(line[2]+line[3])/2
    line=format(line[0],'<10s')+format(line[1],'<10s')+str(line[2])+'
'+str(line[3])+'  '+str(av)+'  '+str(av-cav)+'\n'
    outfile.write(line)
outfile.close()
```

CSV Files

Handling CSV (**Comma-Separated Values**) files in Python is straightforward and can be done using the built-in `csv` module. This module provides methods for reading from and writing to CSV files.

To read data from a CSV file

1. Import the `csv` module.
2. Open the CSV file using the `open()` function.
3. Create a CSV reader object using `csv.reader()` to read data from the file.
4. Iterate through the rows in the CSV file and process the data.

```
import csv

# Open the CSV file for reading
with open("data.csv", mode="r") as file:
   # Create a CSV reader
   csv_reader = csv.reader(file)

   # Iterate through rows and process data
   for row in csv_reader:
      print(row)
```

Here is a sample data file:

```
Name, Age, City
Alice, 30, New York
Bob, 25, Los Angeles
Charlie, 35, Chicago
```

If you are going to process the file you often want to skip the header line:

```
import csv

# Open the CSV file for reading
with open("data.csv", mode="r") as file:
   # Create a CSV reader
   csv_reader = csv.reader(file)

   # Skip the header row (first row)
   next(csv_reader)  # This advances the iterator to the next row

   # Iterate through rows and process data starting from the second row
   for row in csv_reader:
      print(row))
```

To write data to a CSV file

```
import csv

# Data to be written to the CSV file
data = [
   ["Name", "Age", "City"],
   ["Alice", 30, "New York"],
   ["Bob", 25, "Los Angeles"],
]

# Open a CSV file for writing
with open("output.csv", mode="w", newline="") as file:
   # Create a CSV writer
   csv_writer = csv.writer(file)

   # Write data to the CSV file
   csv_writer.writerows(data)
```

How to read a csv file from the document directory (or any other one) in windows.

To read a CSV file from the Documents directory in Windows using Python, you can specify the full path to the file in your code. Here's how you can do it:

- Find the path to the Documents directory on your Windows system. The path is typically located in the user's profile folder. For example, it might be something like `C:\Users\<YourUsername>\Documents`.

- Construct the full path to your CSV file by appending the file's name (including the `.csv` extension) to the Documents directory path.

- Use the full path when opening the CSV file with the `open()` function.

Here's a Python code example that reads a CSV file from the Documents directory:

```
import csv
import os

# Get the path to the Documents directory
documents_directory = os.path.expanduser("~/Documents")

# Specify the CSV file name (change to your file's name)
csv_file_name = "data.csv"

# Construct the full path to the CSV file
csv_file_path = os.path.join(documents_directory, csv_file_name)

# Check if the file exists before opening it
if os.path.exists(csv_file_path):
    # Open the CSV file for reading
    with open(csv_file_path, mode="r") as file:
        # Create a CSV reader
        csv_reader = csv.reader(file)

        # Skip the header row (if needed)
        next(csv_reader)  # This advances the iterator to the next row

        # Iterate through rows and process data
        for row in csv_reader:
            print(row)
else:
    print(f"The CSV file '{csv_file_name}' does not exist in the Documents directory.")
```

Note: "~/Documents" refers to the "Documents" directory in the current user's home directory. "~username/Documents" would refer to the "Documents" directory in the home directory of the user with the username "username." This is portable manner across different systems.

Problem:

Write a program to read the file above and print out the students name followed by their average on all exams.

```
# Define a function to calculate the average score
def calculate_average(scores):
    return sum(scores) / len(scores)

# Open the plain text file for reading
with open("student_scores.txt", mode="r") as file:
    # Iterate through lines in the file
    for line in file:
        # Split the line into words
        words = line.split()

        # Extract the student's name and exam scores
        name = words[0]
        scores = [int(score) for score in words[1:]]

        # Calculate the average score
        average_score = calculate_average(scores)

        # Print the student's name and their average score
        print(f"Student: {name}, Average Score: {average_score:.2f}")
```

and if it’s a CSV file:

```
import csv

def calculate_average(scores):
    return sum(scores) / len(scores)

with open("student_scores.csv", mode="r") as file:
    csv_reader = csv.reader(file)

    # Skip the header row
    next(csv_reader)

    # Iterate through rows and calculate the average score for each student
    for row in csv_reader:
        name = row[0]
        exam_scores = [int(score) for score in row[1:]]
        average_score = calculate_average(exam_scores)

        print(f"Student: {name}, Average Score: {average_score:.2f}")
```