1 Notes

1.1 Variables

```
b = False #Declaration of a false Boolean.
f = 6.67 #Declaration of a Float with value 6.67
s = "I'm a string." #Declaration of a String
l = [1,2,3,'Hello',True] #Declaration of a list.
s = {1,2,3.33,"nice"} #Declaration of a set.
d = {'a':1,'b':2} #Declaration of a dictionary.
t = (45,67) #Declaration of a tuple.
```

1.2 printing

```
inintials = "TF"
print("My initials are:", initials,".") # My initials are: TF.
print(f"My initials are: {initials}.") # My initials are: TF.
print("My initials are: TF.")           # My initials are: TF.
```

1.3 Lists

```
a = [1,2,3,45] #Creates a list with values 1, 2, 3 and 45.
a[1] #Accesses the value in index 1. Here we would get the value 2.
a[-1] #Accesses the value in index -1 which would be the last index. Here we would get the value 45.
a[2:4] #Accesses the values ranging from index 2 to 4. Here: [3,45].
a.append("student") #Adds the new element "student" to the end of the list.
a.remove("student") #Removes the first instance of "student" in the list.
a.index(45) #Finds the index of the first instance of value 45 in the list.
b = ["apple",12]
c = a + b #Concatenates the lists to give: [1,2,3,45,"apple",12].
"apple" in c #Checks if "apple" is in the list c. Here returns True.
len(c) #Returns the length of c. Here it is 6.
set(c) #Returns a list without duplicates, the unique values.
c.insert(1,'koala') # inserts 'koala' at index 1 in list c.
c[::-1] # returns the list in inverted order.
```

1.4 Dictionaries

```
student = {'Math_1': 10,'Programming': 12, 'Chemistry': 4, 'Physics': 10}
student.keys() # returns the keys ['Math_1', 'Programming', 'Chemistry', 'Physics'].
student.values() # returns the values [10, 12, 4, 10].
student['Math_1'] # Accesses the value at key 'Math 1'. Here it returns 10.
student['Biology'] =  7 # Adds a new key with its value.
```

1.5 Strings

```
word = 'hippopotamus' #Definition of a string.
len(word) # Returns the lenght of the word.
word[4] # Accesses the character at index 4. Here it's 'o'.
word[3:7] # Returns the piece of the string ranging from index 3 to 7. Here it's 'popo'.
word.split('o') # Returns a list with cuts of the string at 'o'. Here: ['hipp', 'p', 'tamus'].
word.replace('o','u') # Replaces all instances of 'o' with 'u'. Here: 'hippuputamus'.
number = '1.4'
number.isnumeric() # Checks if the string is a number. Here: True.
number.isdigit() # Checks if the string is an integer. Here: False.
number = float('2.5') # Changes the string to float.
number = int('12') # Changes the string to int.
str_number = str(12) # Changes the int 12 to a string.
word.upper() # Returns the string in all upper case. Here: 'HIPPOPOTAMUS'
word.lower() # Returns the string in all lower case. Here: 'hippopotamus'
word.capitalize() # Returns the string with a capital letter first. Here: 'Hippopotamus'.
word.startswith('hip') # Checks if the string starts with 'hip'. Here: True.
word.endswith('hip') # Checks if the string ends with 'hip'. Here: False.
letter = 'b'
letter.isalpha() # Checks if the character is in the alphabet. Here: True.
letter.isupper() # Checks if the character is in capital. Here: False.
letter.islower() # Checks if the character is in lower case. Here: True.
```

1.6 Conditionals

```
if a>2:
    print(a) #if a is larger than 2, it will print the value of a.
elif a == 2:
    print(a) #if a is not larger than 2 but equal to 2, it will print the value of a.
else:
    print(a) #if a is neither larger than 2 nor equal to 2, it will print the value of a.
```

1.7 Loops

```
for i in range(10):
    print(i) # This for loop will print 0,1,2,3,...7,8,9.

for i in range(1,11):
    print(i) # This for loop will print 1,2,3,...7,8,9,10.

letters = ['a','b','c']
for letter in letters:
    print(letter) #This loop will print a, b, c.

my_number = 0
while my_number<10:
    print(my_number)
    my_number+=1 #This loop will print 0,1,2,3,...,7,8,9,10.
```

1.8 Functions

```
def add_me(a,b):
    return a+b #This function adds a and b and returns their sum.
```

1.9 Classes

```
class Course:
    def __init__(self,name,n_students,n_pass):# __init__() is always ran when a new object is declared.
        self.name = name #self.name is an attribute.
        self.n_students = n_students #self.n_students is an attribute.
        self.n_pass = n_pass #self.n_pass is an attribute.

    def pass_rate(self): #pass_rate is a method
        rate = self.n_pass/self.n_students
        return rate*100

Mathematics = Course('Mathematics',134,89) #Declaring a new object of type Course
Mathematics.pass_rate() #Returns the passing rate of the Math course.
```

1.10 Files

```
pathtxt = 'path/to/file/file.txt'
with open(pathtxt,'r') as file:
    text = file.read() #text will hold all the text as a string

with open(pathtxt,'r') as file:
    textlist = file.readlines() #list where each element is a line from the txt.

with open(pathtxt,'w') as recipe:
    recipe.write('Flour') #creates new file with Flour or overwrites the file.

with open(pathtxt,'a') as recipe:
    recipe.write('\nChocolate')# appends Chocolate in a new line.


import csv
pathcsv = 'path/to/file/file.csv'

with open(pathcsv,'r') as file:
    print(file.read()) #will print all values as one string.

with open(pathcsv,'r') as file:
    A = csv.reader(file)
    for row in A:
        print(row) #will print each element from every row, one by one.

with open(pathcsv,'r') as file:
    A = csv.DictReader(file) # will create dictionaries with keys as the
                             #names of the elements in the first row, and values as the remaining rows.
    for row in A:
        print(row) #print each
```

3 Lists, Dictionaries and Tuples

Lists, Dictionaries and Tuples are very simple and easy to use. One can create lists, dictionaries or tuples of virtually anything (integers, strings, lists, booleans...). The distinction between the three is as follows; lists are the most basic tools for storing data, created by using square brackets. Dictionaries are lists where each value (or element) in the list has a given key (or name). Finally tuples are lists that cannot be modified; in programming terms they are said to be immutable. They are used in rare cases where one doesn't want to modify a list of values.

```
A = [2,4,6,8,10] # integers are full numbers (without decimal)

B = ['chair','table','lamp']

C = [True,"Programming is fun",12,59.667,-3]
```

Listing 1: Creating lists

Here there are 3 different example of lists, the first being with numbers, the second with strings and the third with a mixture of different variable types. Lists are good tools to store, add, remove and edit information.

3.1 List tricks

Here are common commands used when working with lists.

```
D = ['apple','pear','banana']
E = ['dog','cat','bird']

# Second element in D
D[1]

# Tells where 'banana' is in the list
D.index('banana') # returns 2

# adds elements of the list together
K = D+E # K now has 6 strings: ['apple','pear','banana','dog','cat','bird']

# add/remove elements to a list
D.append('grape') # adds 'grape' to D
D.remove('pear') # removes the FIRST instance of 'pear' in the list

# check if an element is in the list
'apple' in D # returns True

# count number of an instance in a list
E.count('dog') # returns 1

# adding words from lists together:
E[0] + ' eats ' + D[2] # returns 'dog eats banana'


# returns length of list:
len(E) # returns 3

set(D) # returns unique elements in list

# inserts an element at specific position
E.insert(1,'koala') # returns ['dog','koala','cat','bird']
```

Listing 2: List basics

3.2 Dictionary tricks

Here are common commands used when working with dictionaries.

```
student = {'Math_1': 10,'Programming': 12, 'Chemistry': 4, 'Physics': 10}

# retrieve keys and values:
student.keys() # returns dict_keys(['Math_1', 'Programming', 'Chemistry', 'Physics'])
student.values() # returns dict_values([10, 12, 4, 10])

# get specific values:
student['Math_1'] # returns 10

# add new keys and values:
student['Biology'] =  7
print(student)
# returns {'Math_1': 10,'Programming': 12, 'Chemistry': 4, 'Physics': 10, 'Biology':7}
```

Listing 3: Dictionary basics

3.3 Tuple tricks

There will typically not be any tuple related questions in the exams. However it is important to know what they are and can be used for. Tuples are essentially like Lists, but where one cannot modify its contents. In a way, a tuple is protected from any modification, and is used for variables that should not be modified in later codes.

4 Strings

Another important part of coding with python is to handle strings. Strings are anything that is related to words and text. Be careful, `'13'` is a string of numbers. In that case the number will be considered as a string, not a number! Here are the basic commands that are useful with strings:

```
word = 'hippopotamus'

# length of word (number of letters)
len(word)
# gives back the 5th letter in the word
word[4] # o
# take a substring of the original
word[3:7] # returns 'popo'

# splits the word into a list, splitting at a specific place in the word
word.split('o') # returns ['hipp', 'p', 'tamus']

# will replace all instances of 'o' with 'u'
word.replace('o','u') # returns 'hippuputamus'

# checks if a word is a number (int)
number = '1.4'
number.isdigit() # returns False (1.4 is not an integer)

# checks if a word is a number (any form, float or int)
number = '1.4'
number.isnumeric() # returns True

# Change the string to a number:
number = float('2.5') # now the number is a float
```

```
# or
number = int('12') # now the number is an integer

#Check if a string contains only alphabetical characters
letter = 'b'
letter.isalpha() # returns True

# makes all letters uppercase
word.upper() # returns 'HIPPOPOTAMUS'

# makes all letters lowercase
word.lower() # returns 'hippopotamus'

# uppercases the first letter in the word
word.capitalize() # returns 'Hippopotamus'


# Check if a letter or string is lowercase, uppercase:
word.capitalize()[0].isupper() # returns True
word.capitalize().isupper() # returns False
word.islower() # returns True

# Check if a string starts or ends with a substring:
word.startswith('hip') # returns True
word.endswith('hip') # returns False
```

Listing 4: String basics

It is important to note that strings differ from lists when calling functions. For example if `a = [1,2,3]` is a list, the function `a.append(4)` adds the integer `4` to the list of values in `a`. However, if another variable `b = 'Hello'` is a string and one tries to call the function `b.capitalize()` it will not change the values of `b`! Instead the function will just show what the values `b` would be if it were capitalized. To change the value of b to its capitalized version, one needs to do `b = b.capitalize()`.

5 If/Elif/Else

In this section the conditionals `if`, `elif` and `else` statements are presented.

`if` statements are used to check if a condition is fulfilled, whether it is a condition with a string, a number, a boolean, a list or something else. Usually one wants to checks whether a number is bigger than another, if a condition is fulfilled, etc. There is an infinite number of possible `if` statements; here are some common examples:

```
A = [1,2,3,4]
D = ['apple','pear','banana']

# checks if the values in A sum up to more than 5:
if sum(A)>5:
    print('The sum of A is higher than 5')

# check if length of A is exactly 4:
if len(A)==4:
    print('the length of the list A is 4')

# check if a list has an 'apple'
if D.count('apple')>=1:
    print('an apple a day keeps the doctor away!')
```

```
# statements can be more complex:

# AND
# check if the length of A is exactly equal to 4 AND sums to larger than 5:
if (len(A)==4 and sum(A)>5):
    print('A is both things')

# OR
# check if the length of A is exactly equal to 4 OR sums to larger than 5:
if (len(A)==5 or sum(A)>5):
    print('A is at least one thing')
```

Listing 5: If Statement

In the examples above the `and` and `or` operators are introduced. As expected, they enable the user to check multiple things at the same time. Let's now look at `elif` and `else` statements.

- `If...Else` is used to check one event, and if that is not the case simply choose the other option:

  'If it's raining outside take your umbrella (specific event), or else leave it at home (all other cases).'

- `If...Elif...Else` is used to check multiple events where only one of the outcomes is possible:

  'Jack wants to know what accessory to bring depending on the weather. If it's raining outside, he should remember his umbrella, or if it is sunny outside, he should take his sunglasses, or else he doesn't need to bring anything.'

This is a very important concept! `if` and `elif` are used when only one of the possible event can happen. The next listing will show the crucial difference between using only `if` statements and using `if/elif` statements.

```
a = 5
# IF/ELIF
if a<2:
    print('a is less than 2')
elif a>3:
    print('a is larger than 3')
elif a>4:
    print('a is larger than 4')
# Will only return one thing: 'a is larger than 3'

# NOT USING ELIF
if a<2:
    print('a is less than 2')
if a>3:
    print('a is larger than 3')
if a>4:
    print('a is larger than 4')
# Will print both: 'a is larger than 3', 'a is larger than 4'

weather = 'sunny'
if weather=='rainy':
    print('remember umbrella')
elif weather=='sunny':
    print('remember sunglasses')
else:
    print('No accessories needed')
# Will print: 'remember sunglasses'
```

Listing 6: If/Elif/Else Statements

The condition that comes after the `if` statement always checks if it is `True` or `False`. If `True`, the code inside the conditional statement will be run, and if it is `False`, it will go and check the next conditional statement.

6 For/While loops

This section is dedicated to the understanding of the `for` and `while` loops. The aim of these fundamental tools in python is to loop over a part of code for a certain number of iterations. The notable difference between the two is that a `for` loop is used when the number of times the code should be iterated is known (fx: running the code 10 times), whereas a `while` loop is sollicited when the number of runs to be carried out is unknown (fx: rolling a dice until it gives the value 6). Here are some examples of both:

```
# Over 7 days, you collect 5 apples times the number of days you've collected them
apples = 0
for i in range(7): # i goes from 0 to 6!
    apples += 5*i
print(apples) # returns 105
# the first day did not count since i started from 0.

# Instead, choose the range of i:
apples = 0
for i in range(1,8): # i goes from 1 to 7!
    apples += 5*i
print(apples) # returns 140
```

Listing 7: For Loop

`for` loops are also widely used when looking in an list, since their sizes are easily found:

```
A = [1,2,3,4,5,6]
K = [0,0,0,0,0,0]

for i in range(len(A)): # range from 0 to length of list A
    K[i] = A[i]+2*i
print(K) # returns [ 1,  4,  7, 10, 13, 16]

# Looking at which animal is domestic:

animals = ['monkey','dog','giraffe','cat','dolphin']
for animal in animals: # loops over all animals
    if animal == ('dog') or animal == ('cat'): # check if animal is a dog or cat
        print('this is a domestic animal: ',animal)
    else:
        print('this is a wild animal: ',animal)
# Returns:
# this is a wild animal:  monkey
# this is a domestic animal:  dog
# this is a wild animal:  giraffe
# this is a domestic animal:  cat
# this is a wild animal:  dolphin

word = 'mOuNTaiN'
uppercases = 0
for letter in word: # looping over letters in a word:
    if letter == letter.upper(): # check if letter is uppercase
        uppercases += 1
print(uppercases) # returns 4

```

```
# Condensed version of a for loop in a list:
A = [1,2,3,4,5,6,7]
B = [i for i in A if i<4]
# returns B = [1,2,3]

# i goes through A and takes those where i<4
```

Listing 8: For Loops Continued

It is important to note that when you are using `for i in range(n)`, `i` will go from 0 to $n-1$. For example, if we have `for i in range(4)`, `i` will take the values `0, 1, 2, 3`, and not `4`. That because by default if the `range()` gets only one value as an input, it will assume that the range starts from `0`. To avoid this, two values need to be provided for the `range`; a starting value and ending value.

```
k = 1
while k<20:  # condition: k is less than 20
    k = k + k**2 # task: add 1 to k
print(k) # returns k = 42

counter = 0
x = 1 # starting point
while x<1200: # condition
    x = x +x/6 + 5 # task
    counter += 1 # count the number of iterations
print(counter) # returns: 24

# You have a bag with 25 pieces of candy, each day you eat 1 piece
Candy = True
Bag = 25

while Candy: # condition: there is still candy left
    Bag = Bag-1 # eat 1 piece of candy
    if Bag<=0: # if the bag is empty
        Candy = False
print(Bag) # returns 0
```

Listing 9: While Loop

The key concept behind `while` loops is that a task can be performed for an indefinite number of iterations. For example, if a formula needs to be calculated recursively until it reaches a limit, but one doesn't know how many times it will take to reach that limit, `while` loops are used.

In general, `while` loops are less controllable statements since they do not need a number of iterations to function. However, `for` loops are used in cases where the number of iterations is known, which is very useful when working with lists or strings.

7 Functions

Functions are ways to organize parts of the code that performs a specific task into one line of code. a function takes some **inputs** and can give back **outputs**. Here is an example of a function:

```
# function that adds a and b together
def addition(a,b):
    x = a + b
    return x
addition(a = 3,b = 4) # returns 7
```

Listing 10: Function Example

In the above the function is **defined** with `def` and the **name** of the function is `addition`. The inputs in the function are `a` and `b`, and the output is `x`. Everything that lies in between is code that will be executed when the function `addition` is used!

Functions are thus building blocks used to run sections of code. In that sense they are quite simple, and thus don't need a lot more explanation. One important aspect however with function is to differentiate between `return` and `print`. In fact, the `return` component gives something back from the function and terminates it, while `print` only informs the user by printing a message when the function is executed. In this case, the program continues (does not stop like a `return`).

```
# function that adds a and b together
def addition(a,b):
    x = a + b
    return(x)

a_plus_b = addition(a = 3,b = 4) # returns 7
type(a_plus_b) # int variable with value 7


def multiplication(a,b):
    x = a * b
    print(x)

a_times_b = multiplication(a = 3,b = 4) # prints 12
type(a_times_b) # NoneType variable
```

Listing 11: Difference between `return()` and `print()`

In the examples above, the `a_plus_b` variable can be used further in other parts of the code, whereas `a_times_b` cannot.

8 Classes

This section will focus on the Object Oriented Programming (OOP) part of python, namely Classes. Classes are extremely useful because we can use them to create objects (which is why it is call [Object]OP) that have a specific number of attributes, properties, functions that they can use.

Taking the example of a string. Once you have defined the variable `a = "Hello World"`, this is equivalent in python to defining an object `a` from the Class `string` whose value is `Hello World`! Now that you have your object `a`, you can use all sorts of functions called `methods` (fancy word for functions inside a class) which belong to the Class `string`. Are you following? If not, read the above a second time and look at the following example:

```
class Course:
    def __init__(self,name,n_students,n_pass):
        self.name = name
        self.n_students = n_students
        self.n_pass = n_pass

    def pass_rate(self):
        rate = self.n_pass/self.n_students
        return rate*100

Mathematics = Course('Mathematics',134,89)

print(Mathematics.name) # returns 'Mathematics'

print(Mathematics.pass_rate()) # returns 66.42
```

Listing 12: Example of a Class

Now in the above there are several things to mention. First the `__init__()` function always needs to be defined when initializing the class to define the classes attributes! What can we know about the class? The `__init__()` is run as soon as one creates an object from that Class, and has to be included when creating it.

Second, the `self` must be used when initializing the attributes of the class, and when calling a method (fancy function). This is essential when building classes.

Finally lets summarize. We have now built a class `Course` with the attributes `name`, `n_students` and `n_pass`. It also has the method `pass_rate` which calculates and returns the percentage of students who passed.

We then created an object `Mathematics` from the class `Courses`. In python-language one would say that the object `Mathematics` is an instance of the Class `Course`. This means that the object mathematics also has the attributes and methods that were defined in the class.

Now that the concept of classes has been introduced, there are some more technical subjects related to them that we will look into. The first is the principle of inheritance. From the example above, one can create a Class, and from there create an Object which is an instance of the Class. What that means is that the Object has the attributes (`self.attribute`) and methods (`def method(self):`) that were defined in the Class.

Similarly, classes can also inherit the attributes and methods from another Class, you could call it a Child Class that inherits attributes from a Parent Class. To do so, when initializing the Child class, the attribute `super().__init__('inputs')` needs to be included. Lets see an example.

```python
class Product:  #PARENT CLASS
    def __init__(self, name, section):
        self.name = name
        self.section = section

    def find_product(self):
        return f'The {self.name} item can be found in {self.section} section.'

banana = Product('banana','fruits&vegetables')
banana.find_product()

# New class who INHERITS the class Product!
class Banana(Product): # CHILD CLASS
    def __init__(self,name, section, ripe):
        super().__init__(name, section) # /!\important line/!\
        self.ripe = ripe

    def is_it_ripe(self):
        if self.ripe:
            return f'the {self.name} is ready to be eaten!'
        else:
            return f'wait a little longer before buying the {self.name}...'

super_banana = Banana('banana','fruits&vegetables',False)
print(super_banana.is_it_ripe())
# prints wait a little longer before buying the banana...

# These can also be used since Banana inherited from the Product class!
print(super_banana.name)
print(super_banana.find_product())
```

Listing 13: Class Inheritance

```python
def __init__(self): # initializion for the class with all attributes
def __str__(self): # when using str(class) or print(class), will run this function
def __add__(self): # when using class1 + class2, will run this function
def __mul__(self): # when using class1 * class2, will run this function
```

Listing 14: Some Class specific methods

The class specific methods are used to change the way simple operators and function work such as strings , $+$, $-$, $*$ operators, and many more. There is an entire world of possibilities with Classes in python, and OOP provides the building blocks to create many fun and interesting projects. However in the scope of this course these are all the aspects of Classes that need to be known.

9 Files

When working in python, one will usually want to use the computational tools available to analyse data, text, images, etc. The most common formats in which such data is saved are either in .csv files, or .txt files.

- .txt files are very simple and store everything as plain text information. It is mostly used to save textual data such as sentences and code.
- .csv files stand for Comma-Separated Values files, and are great for storing numerical data, since they can be split up and categorized with commas. Each line represents a row in a table, and the values in each row are separated by commas to make columns.

To load such a file python needs to know the path to where the file is located, in order to access it from the python script.

```
pathtxt = 'path/to/file/file.txt'

# file.txt
# Bread
# Eggs
# Milk
# Sugar
# Cinnamon

with open(pathtxt,'r') as file:
    text = file.read()
# text is a string with 'Bread\nEggs\nMilk\nSugar\nCinnamon'

with open(pathtxt,'r') as file:
    textlist = file.readlines()
# textlist is a list with ['Bread\n', 'Eggs\n', 'Milk\n', 'Sugar\n', 'Cinnamon']
```

Listing 15: Opening a file

Here the `open()` function takes two inputs: the path and a string. The most common strings used are:

- `'r'` open the file for reading. This string is used when the file should be loaded an read in python.
- `'w'` open the file for writing. Will create the file in the given path, or overwrite an already existing one.
- `'a'` open the file for appending. Will append additional text to the file.

It is important to note that these can be used for **BOTH** .txt and .csv files. Lets look at some examples:

```
pathtxt = 'path/to/file/new_recipe.txt'

with open(pathtxt,'w') as recipe:
    recipe.write('Flour')
    recipe.write('\nEggs')
# This will create a new_recipe.txt file with:
#Flour
#Egg
...
with open(pathtxt,'w') as recipe:
    recipe.write('Chocolate')
# This will overwrite everything in new_recipe.txt!
```

Listing 16: Writing a file

In this case, whenever `'w'` is used, a new file will be created, erasing all the previous content if there was any already there! In order to avoid this, the alternative `'a'` can be used to **append** to a file.

Note that when the `'w'` or `'a'` option is used, this is mostly to save information to a file. As such, they are not readable objects.

```
pathtxt = 'path/to/file/new_recipe.txt'

with open(pathtxt,'w') as recipe:
    recipe.write('Flour')
    recipe.write('\nEggs')
    print(recipe.read()) # WILL NOT WORK!
```

Listing 17: A 'w' file cannot be read at the same time!

```
pathtxt = 'path/to/file/new_recipe.txt'

with open(pathtxt,'w') as recipe:
    recipe.write('Flour')
    recipe.write('\nEggs')
# This will create a new_recipe.txt file with:
#Flour
#Egg
...
with open(pathtxt,'a') as recipe:
    recipe.write('\nChocolate')
# new_recipe.txt is kept the same with the new string:
#Flour
#Egg
#Chocolate
```

Listing 18: Writing vs. Appending a file

Did you notice how many times \n was used in the examples? That is because whenever we are working with text, the computer sees everything as one big line with elements. To tell python that there is a newline, the \n sign is used to indicated that an action is taken. In this case python sees this as 'There is now a new line in the file!'. There are many other actions that can be used with '\', but this is not in the scope of an introductory course.

Finally, .csv files need a package to be opened. Here are some examples:

```
import csv
pathcsv = 'path/to/file/file.csv'
# file.csv:
# Mathematics,10
# Programming,12
with open(pathcsv,'r') as file:
    print(file.read())
# This works but will not recognize the ',' as new columns!
# prints 'Mathematics,10\nProgramming,12'

with open(pathcsv,'r') as file:
    A = csv.reader(file)
    for row in A:
        print(row)
# will recognize ',' as a new column, and '\n' indicates a new row.
# prints:
# ['Mathematics','10']
# ['Proframming','12']

# Other interesting methods:
with open(pathcsv,'r') as file:
    A = csv.DictReader(file) # will create dictionaries with keys as the names of the
    elements in the first row, and values as the remaining rows.
    for row in A:
        print(row)
# file.csv:
# Course,Grade
# Mathematics,10
# Programming,12
# returns:
#{'Course':'Mathematics','Grade':'10'}
#{'Course':'Programming','Grade':'12'}
```

Listing 19: Writing a file