Files

Before we get started with files we can say that jupyter has a mechanism to save the contents of a cell directly to a file. To do this simply type `%%writefile NAME.TXT` at the beginning of the cell. Where NAME.TXT is the file name. Let's try it:

In [2]:
```
%%writefile test.txt
aaa
bbb
ccc
```

```
Writing test.txt
```

By executing the above cell, the file `test.txt` gets created. Now let's see how we can open it and edit it with python. I can "read" this file by storing all its contents in a variable:

In [56]:
```
with open('test.txt') as f:
    d = f.read()
```

In [57]:
```
print (d)
```

```
aaa
bbb
ccc
```

In the above `f` is a new type of variable which represents a file. The commands we can do in a variable that represents a file are:

- `f.read()` : Returns the entire file in a string. **Caution!** Unsuitable for very large files!
- `f.readline()` : Returns the next line of the file.
- `f.readlines()` : Allows a list of all lines in the file. (To be precise with all subsequent lines of the file.

NOTE: Here we are using the letter `f` as an example of a variable name that represents a file. Of course any suitable variable name can be used instead.

In [3]:
```
with open('test.txt') as f:
    d = f.readline()

print (d) # The first line of the file
```

```
aaa
```

In [4]:
```
with open('test.txt') as f:
    d = f.readlines()

print (d) # A list with all lines of the file
```

```
['aaa\n', 'bbb\n', 'ccc\n']
```

Notice that each line ends with a new line (`\n`).

The variable we use in `with` (in our example `f`) can be used in a `for` syntax . Or else we can perform an iteration in a file. We call variables that have this property "itarbles"

(for example list, sets, dictionaries). In the case of a file variable, the for syntax iterates over all the lines of the file:

In [61]:
```
with open('test.txt') as f:
    for line in f:
        print (line)
```

```
aaa

bbb

ccc
```

Notice that `print` also printed the new line character (`\n`) at the end of each line. To remove it we usually do:

In [62]:
```
with open('test.txt') as f:
    for line in f:
        print (line.strip('\n'))
```

```
aaa
bbb
ccc
```

The `with` mechanism is also the "official" way in which python suggests opening files. Another method is the explicit use of the `open` command:

In [3]:
```
f = open('test.txt')
```

After opening it I can read all its contents:

In [4]:
```
a= f.read()
print (a)
```

```
aaa
bbb
ccc
```

If I try to re-read the file then it will return an empty string:

In [5]:
```
f.read()
```

Out[5]:
```
''
```

When we read a python file and get to the end, then python does not continue to read it from the beginning. Instead it returns an empty string.

To re-read a file we must close it:

In [6]:
```
f.close()
```

And to re-open it:

In [7]:
```
f = open('test.txt')
a= f.read()
print (a)
```

```
aaa
bbb
ccc
```

We can read only one line from the file:

In [8]:
```
f = open('test.txt')
line = f.readline()
print (line)
```

```
aaa
```

Notice that the `line` contains the enter (`\n`) at the end of the line:

In [9]:
```
line
```

Out[9]:
```
'aaa\n'
```

If we call `readline` again, then it will read the next line:

In [10]:
```
f.readline()
```

Out[10]:
```
'bbb\n'
```

In [11]:
```
f.readline()
```

Out[11]:
```
'ccc\n'
```

If we reach the end then `f.readline()` returns the empty string:

In [12]:
```
f.readline()
```

Out[12]:
```
''
```

We can also iterate (ie apply `for`) to a file we have opened:

In [13]:
```
f = open('test.txt')
for line in f:
    print (line)
```

```
aaa

bbb

ccc
```

In [15]:
```
f = open('test.txt')
for line in f:
    print (line.strip('\n'))
```

```
aaa
bbb
ccc
```

An (unusual) way to iterate a file is to take advantage of the fact that f.readline () returns the empty string (which evaluates as False):

In [64]:
```
f = open('test.txt')
while True:
    line = f.readline()
    if not line:
        break
    print (line.strip('\n'))
f.close()
```

```
aaa
bbb
ccc
```

The use of `with` instead of the direct use of `open` / `f.close`, however, is also important in the effectiveness of our program. As the official documentation comments, if we forget to call close(), python does NOT guarantee that our data will be saved correctly on the disk!

Creating files

We can create a new file:

In [65]:
```
with open('results.txt', 'w') as f:
    f.write('Hello World\n')
```

In [66]:
```
!cat results.txt # For OSX/Linux
```

```
Hello World
```

In [ ]:
```
!type results.txt # For Windows
```

Notice the `w` in the open command.

Caution!!! If the `results.txt` file already exists then python deletes (overwrites) it without any warning!

But what is this `'w'`? When we open a file we declare to python what we want to do with it. This is called mode. Here is a list of all the modes. The basics are:

- `'r'`; for reading. This is the default mode. When we do not declare any mode (eg `f=open('test.txt')` then it is the same as using `'r'` (i.e. `f=open('test.txt', 'r')`)
- `'w'` for writing. (It overwrites the file if it already exists)
- `'a'` for "append" (ie to continue writing at the end) to a file
- `'x'` for writing in a file if and only if it does not exist. This is particularly usefult if we do not want to risk deleting a file by mistake by using the `'w'` mode.

As with reading, you should make sure to add an enter (`\n`) at the end of a line:

In [5]:
```
# Creating and writing in a new file:
with open('results.txt', 'w') as f:
    f.write('hello')
    f.write('world')
```

In [6]:
```
# Reading from the file i just created:
with open('results.txt') as f:
    data = f.read()
print (data)
```

```
helloworld
```

Notice that if we do not add a new line (`\n'`) all characters are written on the same line!

In [7]:
```
# Creating a file and writing into it:
with open('results.txt', 'w') as f:
    f.write('hello' + '\n')
    f.write('world' + '\n')
```

In [8]:
```
# Reading from the file:
with open('results.txt') as f:
    data = f.read()
print (data)
```

```
hello
world
```

Also the argument in f.write must always be a string:

In [74]:
```
with open('results.txt', 'w') as f:
    f.write(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-74-a433d4705467> in <module>
      1 with open('results.txt', 'w') as f:
----> 2     f.write(10)

TypeError: write() argument must be str, not int
```

Sometimes we may want to write in two separate files. For example suppose we have this list: `a=[1,2,3,4,5]` .

Save in the file `results_1.txt` the multiplication with 2 of each element and in the file `results_2.txt` file the half of each element.

One way is to simply open two files one after the other:

In [76]:
```
a=[1,2,3,4,5]
with open('results_1.txt', 'w') as f:
    for i in a:
        f.write(str(i*2)+'\n')

with open('results_2.txt', 'w') as f:
    for i in a:
        f.write(str(i/2) + '\n')
```

But a more beautiful way is:

In [77]:
```
a=[1,2,3,4,5]
with open('results_1.txt', 'w') as f1, open('results_2.txt', 'w') as f2:
    for i in a:
        f1.write(str(i*2)+'\n')
        f2.write(str(i/2) + '\n')
```

The mechanisms we have learned so far for handling strings can be perfectly combined with reading and writing files. Let's look at some examples:

Example 1

This link contains a link to the `mim2gene.txt` file which is "a tab-delimited file linking MIM numbers with NCBI Gene IDs, Ensembl Gene IDs, and HGNC Approved Gene

Symbols.". Most biology files have this format which is well worth exploring! After downloading the file we can first print the first 10 lines:

In [10]:
```
with open('mim2gene.txt') as f:
    for i, line in enumerate(f):
        print (line.strip('\n'))
        if i >= 10:
            break
```

```
# Copyright (c) 1966-2021 Johns Hopkins University. Use of this file adhere
s to the terms specified at https://omim.org/help/agreement.
# Generated: 2021-03-18
# This file provides links between the genes in OMIM and other gene identif
iers.
# THIS IS NOT A TABLE OF GENE-PHENOTYPE RELATIONSHIPS.
# MIM Number    MIM Entry Type (see FAQ 1.3 at https://omim.org/help/faq)
        Entrez Gene ID (NCBI)   Approved Gene Symbol (HGNC)     Ensembl Gen
e ID (Ensembl)
100050  predominantly phenotypes
100070  phenotype       100329167
100100  phenotype
100200  predominantly phenotypes
100300  phenotype
100500  moved/removed
```

We notice that some lines start with "#" it is common in this way to state that this line contains some commentary instead of some data. However, the 5th row contains the names of each column. Let's export all this!

In [11]:
```
with open('mim2gene.txt') as f:
    for i, line in enumerate(f):

        if i<4:
            continue

        # this is the header!
        header = line
        break

header
```

Out[11]:
```
'# MIM Number\tMIM Entry Type (see FAQ 1.3 at https://omim.org/help/faq)\tE
ntrez Gene ID (NCBI)\tApproved Gene Symbol (HGNC)\tEnsembl Gene ID (Ensemb
l)\n'
```

What are these `\t` in the string? These are tabs. Just as `\n` is a special character to indicate a new line, `\t` indicates a tab. It is very common to use this character to separate columns into files containing multiple columns.

For a start we can remove the `"#"` at the beginning of the header:

In [12]:
```
header = header[1:]
header
```

Out[12]:
```
' MIM Number\tMIM Entry Type (see FAQ 1.3 at https://omim.org/help/faq)\tEn
trez Gene ID (NCBI)\tApproved Gene Symbol (HGNC)\tEnsembl Gene ID (Ensemb
l)\n'
```

Now we can split the header according to the tab character:

In [13]:
```
header = header.split('\t')
header
```

Out[13]:
```
[' MIM Number',
 'MIM Entry Type (see FAQ 1.3 at https://omim.org/help/faq)',
 'Entrez Gene ID (NCBI)',
 'Approved Gene Symbol (HGNC)',
 'Ensembl Gene ID (Ensembl)\n']
```

Hmmm it seems that the first and the last element have extra characters at the beginning and at the end. Let's remove them.

In [14]:
```
header = [x.strip() for x in header]
header
```

Out[14]:
```
['MIM Number',
 'MIM Entry Type (see FAQ 1.3 at https://omim.org/help/faq)',
 'Entrez Gene ID (NCBI)',
 'Approved Gene Symbol (HGNC)',
 'Ensembl Gene ID (Ensembl)']
```

Perfect, we now have a list of file headings. Let's see the other lines:

In [15]:
```
header = None
data = []

with open('mim2gene.txt') as f:
    for i, line in enumerate(f):

        if i<4: # Remove comments
            continue

        if not header:
            # this is the header!
            header = line

            # split and strip the header
            header = header[1:]
            header = header.split('\t')
            header = [x.strip() for x in header]
            continue

        # Here are the rest of the data
        data.append(line.strip('\n').split('\t'))
```

Nice let's see what the first 10 elements of the file are:

In [16]:
```
data[:10]
```

Out[16]:
```
[['100050', 'predominantly phenotypes', '', '', ''],
 ['100070', 'phenotype', '100329167', '', ''],
 ['100100', 'phenotype', '', '', ''],
 ['100200', 'predominantly phenotypes', '', '', ''],
 ['100300', 'phenotype', '', '', ''],
 ['100500', 'moved/removed', '', '', ''],
 ['100600', 'phenotype', '', '', ''],
 ['100640', 'gene', '216', 'ALDH1A1', 'ENSG00000165092'],
 ['100650', 'gene/phenotype', '217', 'ALDH2', 'ENSG00000111275'],
 ['100660', 'gene', '218', 'ALDH3A1', 'ENSG00000108602']]
```

Let's filter out data on which the 5th element is empty:

In [17]:
```
data = [x for x in data if x[4]]
```

We can convert this list into a dictionary. Let's use header as keys!

In [18]:
```
data = [dict(zip(header,x)) for x in data]
```

In [19]:
```
data[100]
```

Out[19]:
```
{'MIM Number': '104615',
 'MIM Entry Type (see FAQ 1.3 at https://omim.org/help/faq)': 'gene',
 'Entrez Gene ID (NCBI)': '6541',
 'Approved Gene Symbol (HGNC)': 'SLC7A1',
 'Ensembl Gene ID (Ensembl)': 'ENSG00000139514'}
```

Finally, let's save the Ensembl Gene ID (Ensembl) and MIM Number columns to a new file:

In [20]:
```
with open('results.txt', 'w') as f:
    # Save header:
    f.write('MIM Number\tEnsembl Gene ID (Ensembl)' + '\n')

    # Save data
    for x in data:
        s = '\t'.join([x['MIM Number'], x['Ensembl Gene ID (Ensembl)']])
        f.write(s + '\n')
```

In [21]:
```
!head results.txt # Prints the first 10 lines of a file in OSX/Linux
```

```
MIM Number      Ensembl Gene ID (Ensembl)
100640  ENSG00000165092
100650  ENSG00000111275
100660  ENSG00000108602
100670  ENSG00000137124
100678  ENSG00000120437
100690  ENSG00000138435
100710  ENSG00000170175
100720  ENSG00000135902
100725  ENSG00000108556
```

In [22]:
```
f=open('results.txt', 'x')
```

```
---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
<ipython-input-22-0b070d8efefb> in <module>
----> 1 f=open('results.txt', 'x')

FileExistsError: [Errno 17] File exists: 'results.txt'
```

Example 2

In this link there is a file with the well known IRIS dataset . Download it and save it in the same directory as python. **CAUTION!** This file does not appear to have the same lines as the original because it contains only 149 data (instead of 150). For the purposes of this example, however, this does not affect us.

The name of the file is: `Wq2PpbZy.txt` .

Let's play with this file!

First we open it:

In [21]:
```
with open('Wq2PpbZy.txt') as f:
    data = f.read()
print(data[:300]) #  Print only the first 300 characters
```

```
Sepal length    Sepal width     Petal length    Petal width     Species
5.2     3.5     1.4     0.2     I. setosa
4.9     3.0     1.4     0.2     I. setosa
4.7     3.2     1.3     0.2     I. setosa
4.6     3.1     1.5     0.2     I. setosa
5.0     3.6     1.4     0.3     I. setosa
5.4     3.9     1.7     0.4     I. setosa
4.6     3.4     1.4     0.3     I. setosa
5.0     3.4     1.5     0.2     I. setos
```

We can get a list of all the lines:

In [22]:
```
data = data.split('\n')
data[:5]
```

Out[22]:
```
['Sepal length \tSepal width \tPetal length \tPetal width \tSpecies',
 '5.2 \t3.5 \t1.4 \t0.2 \tI. setosa',
 '4.9 \t3.0 \t1.4 \t0.2 \tI. setosa',
 '4.7 \t3.2 \t1.3 \t0.2 \tI. setosa',
 '4.6 \t3.1 \t1.5 \t0.2 \tI. setosa']
```

The file appears to be splitting the columns with tabs. Let's split it based on the tabs:

In [23]:
```
data = [x.split('\t') for x in data]
data[:5]
```

Out[23]:
```
[['Sepal length ', 'Sepal width ', 'Petal length ', 'Petal width ', 'Specie
s'],
 ['5.2 ', '3.5 ', '1.4 ', '0.2 ', 'I. setosa'],
 ['4.9 ', '3.0 ', '1.4 ', '0.2 ', 'I. setosa'],
 ['4.7 ', '3.2 ', '1.3 ', '0.2 ', 'I. setosa'],
 ['4.6 ', '3.1 ', '1.5 ', '0.2 ', 'I. setosa']]
```

We notice that there are spaces at the end of some strings. Let's correct this with `strip` :

In [24]:
```
data = [[y.strip() for y in x] for x in data]
data[:5]
```

Out[24]:
```
[['Sepal length', 'Sepal width', 'Petal length', 'Petal width', 'Species'],
 ['5.2', '3.5', '1.4', '0.2', 'I. setosa'],
 ['4.9', '3.0', '1.4', '0.2', 'I. setosa'],
 ['4.7', '3.2', '1.3', '0.2', 'I. setosa'],
 ['4.6', '3.1', '1.5', '0.2', 'I. setosa']]
```

It seems that the first item in the list is the header. Let's take it:

In [25]:
```
header = data[0]
header
```

Out[25]:
```
['Sepal length', 'Sepal width', 'Petal length', 'Petal width', 'Species']
```

It also appears that all other elements (except the first) are data. Let's just take these:

In [26]:
```
data = data[1:]
data[:5]
```

Out[26]:
```
[['5.2', '3.5', '1.4', '0.2', 'I. setosa'],
 ['4.9', '3.0', '1.4', '0.2', 'I. setosa'],
```

```
 ['4.7', '3.2', '1.3', '0.2', 'I. setosa'],
 ['4.6', '3.1', '1.5', '0.2', 'I. setosa'],
```

We notice that the values in the first 4 columns are strings. We can convert them to float:

In [27]:
```
data = [list(map(float, x[:4])) + [x[-1]]  for x in data]
data[:5]
```

Out[27]:
```
[[5.2, 3.5, 1.4, 0.2, 'I. setosa'],
 [4.9, 3.0, 1.4, 0.2, 'I. setosa'],
 [4.7, 3.2, 1.3, 0.2, 'I. setosa'],
 [4.6, 3.1, 1.5, 0.2, 'I. setosa'],
 [5.0, 3.6, 1.4, 0.3, 'I. setosa']]
```

Now I can do a little "browsing" on it. For example, how many total species are there?

In [28]:
```
species = set([x[-1] for x in data])
species
```

Out[28]:
```
{'I. setosa', 'I. versicolor', 'I. virginica'}
```

How much data do we have for each species?

In [29]:
```
b = [(x, sum([y[-1]==x for y in data])) for x in species]
b
```

Out[29]:
```
[('I. versicolor', 50), ('I. virginica', 49), ('I. setosa', 50)]
```

We can also turn this into a dictionary

In [31]:
```
dict(b)
```

Out[31]:
```
{'I. versicolor': 50, 'I. virginica': 49, 'I. setosa': 50}
```

What is the average "Sepal Length" for all data?

In [33]:
```
def average(x):
    return sum(x)/len(x)

average([x[header.index('Sepal length')] for x in data])
```

Out[33]:
```
5.831543624161076
```

What is the average "Sepal length" for each species separately?

In [34]:
```
[(x, average([y[0] for y in data if y[-1] == x])) for x in species]
```

Out[34]:
```
[('I. versicolor', 5.936),
 ('I. virginica', 6.565306122448979),
 ('I. setosa', 5.007999999999999)]
```

How many "I. setosa" have Petal length> 1.5?

In [35]:
```
sum([x[2] > 1.5 and x[-1] == 'I. setosa' for x in data])
```

Out[35]:
```
13
```

Which are these;

In [36]:
```
[x for x in data if x[2] > 1.5 and x[-1] == 'I. setosa']
```

Out[36]:
```
[[5.4, 3.9, 1.7, 0.4, 'I. setosa'],
```

```
 [4.8, 3.4, 1.6, 0.2, 'I. setosa'],
 [5.7, 3.8, 1.7, 0.3, 'I. setosa'],
 [5.4, 3.4, 1.7, 0.2, 'I. setosa'],
 [5.1, 3.3, 1.7, 0.5, 'I. setosa'],
 [4.8, 3.4, 1.9, 0.2, 'I. setosa'],
 [5.0, 3.0, 1.6, 0.2, 'I. setosa'],
 [5.0, 3.4, 1.6, 0.4, 'I. setosa'],
 [4.7, 3.2, 1.6, 0.2, 'I. setosa'],
 [4.8, 3.1, 1.6, 0.2, 'I. setosa'],
 [5.0, 3.5, 1.6, 0.6, 'I. setosa'],
 [5.1, 3.8, 1.9, 0.4, 'I. setosa'],
```

What are the indexes of the above (I. setosa that have Petal length> 1.5)

In [37]:
```
[i for i,x in enumerate(data) if x[2]>1.5 and x[-1] == 'I. setosa']
```

Out[37]:
```
[5, 11, 18, 20, 23, 24, 25, 26, 29, 30, 43, 44, 46]
```

What is the minimum Sepal length for everything?

In [38]:
```
# We will talk about lambda a bit later
min(data, key=lambda x:x[header.index('Sepal length')])
```

Out[38]:
```
[4.3, 3.0, 1.1, 0.1, 'I. setosa']
```

What is the minimum Sepal Length for all different flower types?

In [39]:
```
[min([(y[0], y) for y in data if y[-1] == x]) for x in species]
```

Out[39]:
```
[(4.9, [4.9, 2.4, 3.3, 1.0, 'I. versicolor']),
 (4.9, [4.9, 2.5, 4.5, 1.7, 'I. virginica']),
 (4.3, [4.3, 3.0, 1.1, 0.1, 'I. setosa'])]
```

What is the index of the above?

In [40]:
```
[min([(y[0], y, i) for i,y in enumerate(data) if y[-1] == x]) for x in spe
```

Out[40]:
```
[(4.9, [4.9, 2.4, 3.3, 1.0, 'I. versicolor'], 57),
 (4.9, [4.9, 2.5, 4.5, 1.7, 'I. virginica'], 106),
 (4.3, [4.3, 3.0, 1.1, 0.1, 'I. setosa'], 13)]
```

What is the range (ie the smallest and largest) of "Sepal Length" for each species separately?

In [41]:
```
# Returns the range of a list
def my_range(l):
    return min(l), max(l)

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

Out[41]:
```
(3, 9)
```

In [42]:
```
[ (my_range([y[0] for y in data if y[-1] ==x]), x) for x in species]
```

Out[42]:
```
[((4.9, 7.0), 'I. versicolor'),
 ((4.9, 7.9), 'I. virginica'),
 ((4.3, 5.8), 'I. setosa')]
```

Let's get the indexes of the smaller and larger:

In [43]:
```
# Returns the index of the maximum element
def max_index(l):
    return max(range(len(l)), key=lambda x : l[x])

max_index([3,6,5,8,7])
```

Out[43]:
```
3
```

In [44]:
```
# Get the index of the minimum
def min_index(l):
    return min(range(len(l)), key=lambda x : l[x])

min_index([3,6,5,8,7])
```

Out[44]:
```
0
```

In [45]:
```
def my_range_2(l):
    return 'min:{} min_index:{} max:{} max_index: {}'.format(
        min(l), min_index(l), max(l), max_index(l)
    )
```

In [46]:
```
[ (my_range_2([y[0] for y in data if y[-1] ==x]), x) for x in species]
```

Out[46]:
```
[('min:4.9 min_index:7 max:7.0 max_index: 0', 'I. versicolor'),
 ('min:4.9 min_index:6 max:7.9 max_index: 31', 'I. virginica'),
 ('min:4.3 min_index:13 max:5.8 max_index: 14', 'I. setosa')]
```

Which dataset has the largest sepal area? As area we define the product Sepal width * Sepal length

In [47]:
```
max([(x[0]*x[1], x, i) for i,x in enumerate(data)])
```

Out[47]:
```
(30.02, [7.9, 3.8, 6.4, 2.0, 'I. virginica'], 131)
```

Which dataset has the largest area for each species?

We create a function that takes a list of data and calculates which of them has the largest area:

In [48]:
```
f = lambda d : max([(x[0]*x[1], x, data.index(x)) for x in d])
```

eg: of all the data, the largest area has:

In [49]:
```
f(data)
```

Out[49]:
```
(30.02, [7.9, 3.8, 6.4, 2.0, 'I. virginica'], 131)
```

We apply it for each species separately:

In [50]:
```
[f([y for y in data if y[-1] == x]) for x in species]
```

Out[50]:
```
[(22.400000000000002, [7.0, 3.2, 4.7, 1.4, 'I. versicolor'], 50),
 (30.02, [7.9, 3.8, 6.4, 2.0, 'I. virginica'], 131),
 (25.080000000000002, [5.7, 4.4, 1.5, 0.4, 'I. setosa'], 15)]
```

String formatting

Mixing variables with strings in a fast and easy fashion is one of the reasons that python was created!

So suppose we have the variables:

In [96]:
```
name = "Mitsos"
age = 40
```

And we want to place them on the string: `My name is .... and I am ... years old` . We have the following options:

1. string concatenation

One of the most basic options. It does not give us many possibilities but it is very simple:

In [97]:
```
print ('My name is ' + name + ' and I am ' + str(age) + ' years old')
```

```
My name is Mitsos and I am 40 years old
```

2. Use the format command:

In [98]:
```
print ('My name is {} and I am {} years old'.format(name, age))
```

```
My name is Mitsos and I am 40 years old
```

3. Use the format command with placeholders:

In [102…
```
print ('My name is {NAME} and I am {AGE} years old'.format(NAME=name, =age
```

```
My name is Mitsos and I am 40 years old
```

This allows us to use the same placeholder multiple times

In [105…
```
a = 'James'
b = 'Bond'

print ('My name is {SURNAME}. {NAME} {SURNAME}.'.format(NAME=a, SURNAME=b)
```

```
My name is Bond. James Bond.
```

4. Use the format command for "clever" formatting

The format has a subset of commands to print numbers to with specific accuracy e.g. 3 decimal places:

In [107…
```
a=234/1345
print (a)
print ('the result is {0:.3f} '.format(a))
```

```
0.17397769516728626
the result is 0.174
```

Python essentially supports a mini-language](https://docs.python.org/3/library/string.html#formatspec or else a language within the language to format strings more intelligently. For example: right alignment with a maximum of 30 characters:

In [114…
```
a = 'Hello'
b = 'Heraklion'

print ('{0:>30}'.format(a))
print ('{0:>30}'.format(b))
```

```
                         Hello
                     Heraklion
```

5. The `%` operator

Instead of the format we can use the `%` operator. This is no longer common and is more of a remnant of python 2 :

In [118…
```
print ('My name is %s and I am %s years old' % (name, age))
```

```
My name is Mitsos and I am 40 years old
```

6. f strings

f strings are the most modern method of formatting strings together with variables. The f strings differ from other methods in that the replacement takes place during string creation. That is, all other methods (except the 1st) first make the string without the variables in memory and then replace the variables where needed. As a consequence it is extremely fast. An f-string is declared by prefixing with the character `f` any string. Inside the string you can user any variables (and expressions) directly:

In [120…
```
print (f'My name is {name} and I am {age} years old')
```

```
My name is Mitsos and I am 40 years old
```

pass

In any new indentation (for, while, def, if, ...) we must have at least one command:

In [121…
```
if True:
```

```
  File "<ipython-input-121-2c8a33b52dfb>", line 1
    if True:
            ^
SyntaxError: unexpected EOF while parsing
```

If for some reason we do not want to put any command, then we can use the `pass` :

In [122…
```
if True:
    pass
```

In [123…
```
def f():
    pass
```

We can use `pass` anywhere and it does not do anything :

In [124…
```
print ('Hello')
pass
print ('World')
```

```
Hello
World
```

The operator `is`

We have seen many times the operator `is` . We used it to see what type a variable is:

In [125…
```
a = [1,2,3]
type(a) is list
```

Out[125… `True`

In [126…
```
type(a) is str
```

Out[126…
```
False
```

The operator `is` looks at whether two variables are the same. Or look at whether they refer to the same memory location. This is different from equal: Two variables are *equal* (`==`) if they have the same value. Two variables are the *same* if they refer to the same memory location. Examples:

In [127…
```
a = [1,2,3]
b = [1,2,3]
print (a==b)
print (a is b)
```

```
True
False
```

In [128…
```
a = [1,2,3]
b = a
print (a==b)
print (a is b)
```

```
True
True
```

When two variables are the same (and not just equal) then if you change one, the other also changes:

In [129…
```
a = [1,2,3]
b=a
b[0] = 100
print (a)
```

```
[100, 2, 3]
```

Another example:

In [130…
```
a = [1,2,3,4]
b = [a,a,a]
print (b)
a[0]=100
print (b)
```

```
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
[[100, 2, 3, 4], [100, 2, 3, 4], [100, 2, 3, 4]]
```

We notice that we changed only the first element of the first list (`a[0]=100`) and yet they all changed! This is because all the values in the list are the same variable `b = [a,a,a]` !

This creates the following strange phenomenon:

In [143…
```
a = [[]] * 3 # The inner list is the same for all three elements of the "ou
print (a)
```

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

In [144…
```
a[0].append('mitsos')
print (a)
```

```
[['mitsos'], ['mitsos'], ['mitsos']]
```

The above does not apply to primitive types, (ie int, str, float, bool, complex):

In [134…
```
a = 1000
b = 1000
a is b
```

Out[134…
```
False
```

In [135…
```
a = 1000
b = 1000
a=b
a is b
```

Out[135…
```
True
```

Although they are the same, if I change `b` then `a` does NOT change:

In [136…
```
b = b + 1
a is b
```

Out[136…
```
False
```

That is, every time I change the value of a variable, its memory location also changes. To see what is the code of the memory location of a variable we use the `id` :

In [140…
```
a = 1
id(a)
```

Out[140…
```
4398221664
```

In [141…
```
a = a + 1
id(a) # The position changed!
```

Out[141…
```
4398221696
```

In [142…
```
a = [1,2,3]
b = a
print (id(a))
print (id(b)) # The position didn't change!
```

```
140652702590144
140652702590144
```

Ternary operator

With the ternary operator we can write an `if ... else ...` syntanx as an expression. The structure is:

```
a = EXPRESSION_IF_THE_CONDITION_IS_TRUE if CONDITION else
EXPRESSION_IF_THE_CONDITION_IS_FALSE
```

For example:

In [147…
```
age = 40
if age>=18:
    status = 'adult'
else:
    status = 'not adult'
print (status)
```

```
adult
```

The same with ternary operator:

In [148…
```
status = "adult" if age>=18 else "not adult"
print (status)
```

```
adult
```

In [145…
```
a = 1 if 5<3 else 8
print (a)
```

```
8
```

In [146…
```
a = 1 if 3<5 else 8
print (a)
```

```
1
```

We can compose many ternary operators together:

In [149…
```
a = (1 if 3<5 else 6) if 5>6 else (6 if 4>1 else 7)
print (a)
```

```
6
```

If I take out the parenthesis `a` will have a different value. (Why;)

In [150…
```
a = 1 if 3<5 else 6 if 5>6 else 6 if 4>1 else 7
print (a)
```

```
1
```

lambda functions

Lambda functions are special functions that have the following properties:

- They have no name .
- They only contain a single expression which is the return value of the function. That is, they can not have more than one line.

In [152…
```
f = lambda x : x/2
f(10)
```

Out[152…
```
5.0
```

The above is equivalent to:

In [154…
```
def f(x):
    return x/2
f(10)
```

Out[154…
```
5.0
```

Note that we can completely omit the letter `f` which is the name of the function:

In [155…
```
(lambda x : x/2)(10)
```

Out[155…
```
5.0
```

Lambda functions have exactly the same type as "normal" functions:

In [156…
```
type(lambda x:x)
```

Out[156… function

Why are lambda functions useful? Many times we need to use a function that does something simple or we are going to use it once, so there is no reason to name it as a variable. This often happens when we want to give a function as an argument to another function, or to functions that return functions, or to functions that are in lists and dictionaries.

For example:

In [23]:
```
a = {
    'accurate': lambda x:x/3, # Division with 3
    'not_accurate': lambda x:x//3 # Integer division with 3
}
print(a['accurate'](55))
print (a['not_accurate'](55))
```

```
18.333333333333332
18
```

However, where the lambda functions are more commonly used are in the `map`, `filter`, `min`, `max`, `sorted` functions. For example, let the following list:

In [158…
```
a = [1,2,3,4,5,6,7,8,9,10]
```

Multiply all the data by 2:

In [159…
```
def f(x):
    return x*2
print (list(map(f, a)))
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

In [160…
```
print (list(map(lambda x:x*2, a)))
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Get only odd numbers:

In [161…
```
def f(x):
    return x%2==1
print (list(filter(f, a)))
```

```
[1, 3, 5, 7, 9]
```

In [162…
```
print (list(filter(lambda x:x%2==1, a)))
```

```
[1, 3, 5, 7, 9]
```

Let the list be:

In [166…
```
cities = ['edessa', 'bethleem', 'Heraklion', ]
```

Sort them according to the occurence of the letter "e":

In [167…
```
def f(x):
    return x.count('e')
print (sorted(cities, key=f))
```

```
['Heraklion', 'edessa', 'bethleem']
```

In [168…
```
print (sorted(cities, key=lambda x: x.count('e')))
```

```
['Heraklion', 'edessa', 'bethleem']
```

Let the following list:

In [170…
```
cities = [
    ('Heraklion', 200_000),
    ('Athens', 1_000_000),
    ('Thessaloniki', 500_000),
]
```

What is the city with the most inhabitants?

In [171…
```
def f(x):
    return x[1]

max(cities, key=f)
```

Out[171…
```
('Athens', 1000000)
```

In [172…
```
max(cities, key=lambda x:x[1])
```

Out[172…
```
('Athens', 1000000)
```

Variable scoping or Variable visibility

By this strange term we refer to the "scope" of variables in a programming language. By scope we mean the functions and structures from which we can access a variable. More: https://en.wikipedia.org/wiki/Scope_%28computer_science%29

We start by saying that a variable that is defined outside the function can be accessed inside the function (this is something we generally try to avoid).

In [173…
```
a = 3

def f():
    print (a)

f()
```

```
3
```

However, if a variable is **defined** within a function then it becomes "independent". That is, it is defined for the function and the function only:

In [174…
```
a = 3
def f():
    a=5

f()
print (a)
```

```
3
```

We observe that although we set `a=5` inside the function, the `a` variable outside the function did not change. This is because the two variables, although having the same name, are different.

One possible question is: how is it that when I `print (a)` in the 1st example I refer to "outside" `a` and when I use `a = 5` in the 2nd example, I refer to "inside" a? This is

quite confusing but python has this principle: if you put a new value in a variable anywhere in a function then you redefine it. For example:

In [176…
```
a=5
def f():
    print (a)
    a=5

f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-176-835a5d405592> in <module>
      4     a=5
      5
----> 6 f()

<ipython-input-176-835a5d405592> in f()
      1 a=5
      2 def f():
----> 3     print (a)
      4     a=5
      5

UnboundLocalError: local variable 'a' referenced before assignment
```

What happened here; Why this error showed up? As we said before: if you put a new value in a variable anywhere in a function then you redefine it. So a was redefined in the function. So when we go to print (a), then we tell python to print the variable a of the function which .. has not been defined yet!

If for some reason we want to say that a in f refers to "outside" a and not to the "inside" we must explicitly state it:

In [178…
```
a=5
def f():
    global a

    print (a)
    a=6

f()
print (a)
```

```
5
6
```

Here we observe the following: we changed a inside the function and it changed outside as well!

It is also obvious that a variable defined inside a function does not "exist" outside of it:

In [182…
```
def f(x):
    t = "hello"
print (t)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-182-4a4591587f68> in <module>
      1 def f(x):
      2     t = "hello"
```

```
----> 3 print (t)
```

What about the parameters of the functions? If we change a parameter inside the function, will it change outside as well? The answer is .. it depends! If the parameter is int, float, string, bool, complex, None (or otherwise primitive data types). Then it does not change:

In [180…

```
def f(x):
    x += " mitsos"

x = "My name is:"
f(x)
print (x)
```

```
My name is:
```

But if it is list, dictionary, set, class (we will talk about them later). Then it changes!

In [181…

```
def f(l):
    l.append(5)

a = [1,2,3,4]
f(a)
print (a)
```

```
[1, 2, 3, 4, 5]
```