MLJC UniTo

4th Workshop

Welcome to our first lecture of 2020, in the next lines we wrote down some micro-examples to review what has been done until now. Up to here you should be at least familiar with:

- The simplest terminal commands of your operating system (we are using Windows 10 on this machine): how to navigate directories, display their contents, create files and directories. (cd, pwd, ls, mkdir, ecc. for Windows)
- Commands meant to open files through programs of your choice, install new programs and modules through the terminal
- The logic of a Jupyter Notebook, the good practice of writing neat and commented code
- The importance of operating in dedicated ENVIRONMENTs for each project to avoid dependency issues
- Doing simple math with Python and the math module
- How Python handles types
- Conditional Operators (if, elif, else)
- Iterations (for, while)
- Strings and related methods
- Dictionaries, Sets, Lists, Tuples
- Mutable and Immutable objects
- I/O from/to file

In [262]:

```
cd \Users\matteo
```

```
C:\Users\matteo
```

In [539]:

```
# We changed directory with cd.
# DO NOT comment code referred to the command line in the same cell of your notebook, it w
ill yield an error
# Python works diligently to streamline your commands in the right place, based on context
and syntax,
# but the Windows Shell interpreter has a different syntax for comments and inteprets them
as commands instead.
```

In [264]:

```
pwd
```

Out[264]:

```
'C:\\Users\\matteo'
```

In [265]:

```
# pwd shows us the present working directory. Modules or files we wish to import should be in it, or we should explicitly refer to their path,
# but this is inconvenient for a huge number of reasons
# Your pwd should be your workspace, tidy and with everything you need in it, not a bit more.
# Same logic as with environments: any version of any project gets its own private directory/environment (in an ideal world)
```

In [266]:

```
cd \Users\matteo\MLJC
```

```
C:\Users\matteo\MLJC
```

In [267]:

```
cd \Users\matteo\MLJC\Lecture04
```

```
C:\Users\matteo\MLJC\Lecture04
```

In [268]:

```
pwd
```

Out[268]:

```
'C:\\Users\\matteo\\MLJC\\Lecture04'
```

In [269]:

```
ls
```

```
 Il volume nell'unità C è SYSTEM
 Numero di serie del volume: 72D7-2414

 Directory di C:\Users\matteo\MLJC\Lecture04

22/01/2020  13:13    <DIR>          .
22/01/2020  13:13    <DIR>          ..
22/01/2020  11:14    <DIR>          .ipynb_checkpoints
21/01/2020  14:44    <DIR>          addutils
21/01/2020  14:44    <DIR>          example_data
21/01/2020  11:46    <DIR>          HelloWorld
21/01/2020  14:44    <DIR>          images
22/01/2020  13:13            25.585 Lecture04.ipynb
21/01/2020  14:22             7.076 Lecture04-checkpoint.ipynb
27/11/2019  19:28           168.120 py01v04_ipython_notebook_introduction.ipy
nb
27/11/2019  19:10            55.499 py02v04_python_basics.ipynb
22/01/2020  11:12            36.748 py03v04_python_getting_started.ipynb
27/11/2019  19:10            29.078 py04v04_python_style_guide.ipynb
22/01/2020  11:12            25.308 py05v04_python_more_examples.ipynb
22/01/2020  11:12            31.656 py06v04_python_object_oriented.ipynb
27/11/2019  19:10            42.099 py07v04_Unicode.ipynb
22/01/2020  11:12            28.477 py08v04_python_regular_expressions.ipynb
22/01/2020  11:16            49.451 py09v04_ipython_notebook_widgets.ipynb
27/11/2019  19:10             2.234 requirements.txt
21/01/2020  14:44    <DIR>          SummerCoding-master
21/01/2020  14:44    <DIR>          tmp
22/01/2020  11:35        49.348.076 UFOSightings.xlsx
22/01/2020  11:56       119.872.468 UFOSightingsCSV.csv
21/01/2020  14:44    <DIR>          utilities
              14 File    169.721.875 byte
              10 Directory  147.881.177.088 byte disponibili
```

In [270]:

```
# With ls we print the contents of the folder and some info on the system/folder
# All these commands have optional parameters that I suggest you look into, even if it's n
ot prioritary for our course
# Working with computers "the old way" (without a graphic user interface) lets you experie
nce the inner workings of the machine in a unique fashion
```

In [271]:

```
mkdir \Users\matteo\MLJC\Lecture04\HelloWorld
```

```
Sottodirectory o file \Users\matteo\MLJC\Lecture04\HelloWorld già esistente.
```

In [540]:

```
# With mkdir we create a new directory
```

In [273]:

```
# We define a trivial function
# Notice the indenting (number of spaces, Python uses 4) after the column (:)
# Python doesn't work with brackets {[()]}: it interprets indentation as dependency

def f(x):
    print(x)
```

In [274]:

```
f(2)
```

2

In [275]:

```
# A lambda function is an anonymous function. It doesn't need a name or definition but can
be assigned to a variable and work like a function
# It accepts only one input (e.g. it can be a list, but only one) and spouts only one outp
ut
# A lambda function can only define SINGLE LINE EXPRESSIONS
# Computationally faster than a regular function
# The concept of these simple maps is derived from Lambda Calculus, a logical abstraction
 that models computation.

g = lambda x : print(x)
```

In [276]:

```
g(2)
```

2

In [542]:

```
# We define a simple function that accepts raw input (if this was a script being executed,
it would accept it through the terminal)

def acquire_variable() :
    _ = input() # converts all input to a single string, then we get it back as a number w
ith float()
    _ = float(_)
    return _
```

In [280]:

```
variable = acquire_variable()
```

In [281]:

```
variable
```

Out[281]:

```
47.0
```

In [282]:

```
# We define a mindless conditional to see how if, elif, else work together to construct de
cision trees

def simple_conditional() :

    _ = input()

    if (len(_) < 7) == True :

        print('Talk more')

    elif (len(_) > 7) == True :

        print('Shut up you blubbermouth')

    else :

        print('')
        print ('The number seven is my dearest. Thank you, kind stranger :D')
```

In [283]:

```
simple_conditional()
```

```
The number seven is my dearest. Thank you, kind stranger :D
```

In [284]:

```
# We import the modules provided from AddFor

import addutils.toc ; addutils.toc.js(ipy_notebook=True)
```

Out[284]:

We haven't reviewed all of what we've seen, just the very basics to grasp what's going on, but don't panic: in the next section we develop a long exercise that calls on all the instruments you got to know and some more we'll introduce on the go, let's get our hands dirty :).

- We will clean and catalogue our data
- We will try to visualize some stats

The UFO sightings dataset provided by a US organization will be ours to dissect today: you can download it from http://bit.ly/UFOrobot (http://bit.ly/UFOrobot)

To work with such a vast dataset we'll need to upload it into our program: instead of creating lists for all columns, or a table representing the data, we want to store each sighting in a dedicated object, holding all the relevant attributes. This is called a class.

Classes

When programming, we usually work with structured data (i.e. not mere numbers, or strings, but collections of various *media*) and often we want to create our own, defining specific methods (functions) that work only for that type of data. In Python this is achieved through classes.

Let's say we are out to get those reptilian aliens everyone is talking about, we should know where they do their business, the U.S. state they indulge in the most, at least. We also want to know in how many shapes they come and how frequently. So to get a tidy database in our number-cruncher let us create a class:

In [337]:

```
# A class is a structure for a collection of attributes and methods on those attributes, a
ll accessible through the .(dot) suffix
# The text enclosed by """ is the documentation written for the class.
# If present, you can access it for any (built-in or not) class with class_name.__doc__

class Sighting:
    """A class that refers to UFO sightings by source of data, US city and state, date, sh
ape observed, duration"""
    source = 'National UFO Research Center' # This attribute is shared by all instances, s
o it is placed before the initialization

    # We can re-define standard methods for classes: __init__ is called any time an object
is created,
    # here we tell the interpreter that anytime an instance of the class Sighting is evalu
ated,
    # the standard initialization must be overwritten by the Class definition of this meth
od.
    # Python's classes are more flexible, albeit more confusing than in C#,

    def __init__(self, city, state, date, time, shape):
        self.city = city
        self.state = state
        self.date = date
        self.time = time
        self.shape = shape

    def print_specifics(self):
        print("I saw a %s in %s on %s, at %s" % (self.shape, self.state, self.date, self.t
ime))
```

In [338]:

```
Test = Sighting('Dallas','TX', 'April 1 2018', '00:00', 'cigar')

print(Test.shape)
Test.print_specifics()
print(Test.__doc__)
```

```
cigar
I saw a cigar in TX on April 1 2018, at 00:00
A class that refers to UFO sightings by source of data, US city and state, da
te, shape observed, duration
```

Subclasses

We are some quite skeptical ufologists and want to create a subclass of "Unreliable sightings" to distinguish the highly certified sightings of our verified members from drunk rednecks seeing a shooting star.

This subclass should **inherit the structure** of Sightings and add a slot for reasons to doubt that sighting (drunk, psychotic, on drugs, on medications, visually impaired).

When we are digging deep in the data, looking for connections between UFO sightings and pyramid schemes, we also want a warning to pop when we invoke the `.print_specifics` method: this sighting is considered just for the sake of knowledge, but is deemed unreliable.

To do so, we must overwrite the class-specific method and create a subclass-specific method.

We could achieve this through another attribute and an *if* statement, but we don't want to burden normal instances of the class Sightings with an unnecessary check.

In [543]:

```
# the *warnings is a Python syntax to indicate any other number of arguments, the collecti
on of them is named warnings, but can take any name:
# we can iterate on these optional arguments as on a set
# A double asterisk (**) indicates any number of dictionaries as potential variables (keyw
ord variables)
# This is done to avoid input errors or to compute functions that operate on sizeable coll
ections rather than a fixed number of objects.
# It is not advisable to use * and ** when a large number of variable is involved
# The * and ** operators have a nice and intuitive way of working polymorphically, we'll c
heck it out later, roughly they pack/unpack serial objects
# (POLYMORPHISM = working on different types/contexts w/out the need to specify them)

class Unreliable_Sighting(Sighting):
    """A subclass of the class Sighting that adds /the altered state of consciousness at s
ighting/ and /potential warnings/ to the Mother class"""
    def __init__(self, city, state, date, time, shape, altered_state, *warnings):
        super().__init__(city, state, date, time, shape)
        self.altered_state = altered_state
        self.warning = warnings

# The super() function grants access to the SuperClass of an object, i.e. the mother class
# The super() syntax is a bit different between Python3.x and Python2.x
# We are using Python3, the notebooks from AddFor are written for Python2, so refer to the
m (or anywhere on the internet) for the syntax

    def print_specifics(self):
        print("I saw a %s in %s on %s, at %s, but I was %s" % (self.shape, self.state, sel
f.date, self.time, self.altered_state))
        print("WARNING: Unreliable Sighting. Only trust NUFORC approved aliens")

        for note in self.warning:
            print('WARNING: %s' % note)
```

In [316]:

```
Test = Unreliable_Sighting('Dallas', 'TX', 'April 1 2018', '00:00', 'cigar','on LSD','The
 subject was exposed to classified information during the Vietnam War')

Test.print_specifics()
```

```
I saw a cigar in TX on April 1 2018, at 00:00, but I was on LSD
WARNING: Unreliable Sighting. Only trust NUFORC approved aliens
WARNING: The subject was exposed to classified information during the Vietnam
War
```

Reading from file

Now that our UFO Sighting class is in place, we'd want a list of sightings to act on. Go to https://bit.ly/UFOrobot (https://bit.ly/UFOrobot) and download the data. It is in csv format, so open excel or your favorite table editor and tidy it up a bit. We want to keep only city, state, date, time and shape. We're not going to use our Unreliable_Sighting class for now, unless some of you are on drugs.

We want to keep only data that has a True value (non Null, in Python when evaluated as a boolean any variable type that is non empty/non trivial is True, zero or empty is False) for all of our entries. So we'll have to run a conditional at the start of our acquisition.

When reading from file, we usually build a list from it to avoid opening and closing it everytime.

We assign the path to a path variable, then call the open() function: this is the main way of working with files.

`open()` accepts a path in string format and a parameter from this list, differentiating ways of summoning a file into Python

'r' open for reading (default)

'w' open for writing, truncating the file first

'x' create a new file and open it for writing

'a' open for writing, appending to the end of the file if it exists

'b' binary mode

't' text mode (default)

'+' open a disk file for updating (reading and writing)

Any file format has (usually more than) one dedicated library/module to be handled with. We'll use `csv` for now. Were we to use xlsx files we'd use `import xlsxwriter` .

As a .csv is not a plain-text file (.txt *et al.*), the `open()` object has no way to access or loop over the table's elements, that is why we need to include `import csv` in our header: it's a Python cookbook for .csv files.

To link together the operation of passing the file to an internal variable and then parse it into the csv class we use:

```
with open('File_Name', 'r') as CSV_File:
       Read_CSV_File = csv.reader(CSV_File, delimiter=',')
```

Where the `with` method compiles like

```
File = open('File_Name','r')
Read_File = csv.reader(File, delimiter=',')
```

but with better exception handling and a protocol that always closes files after using them. The `as` statement just creates an alias for the `open()` object, valid only inside the `with` statement.

In [544]:

```
# We import the library to handle .csv files, then we create the file variable with open
().
# The file needs to be in a format that is useful within the Python framework:
# We use the builtin methods of the csv module

import csv

with open('UFO_Sightings_Clean.csv', 'r') as Raw_CSV_File:
    Read_CSV_File = csv.reader(Raw_CSV_File, delimiter=';')

    _ = 0
    for row in Read_CSV_File:
        if _ == 0:
            print('-These should be our chosen categories:')
            for i in range(4):
                print(row[i])
            _ += 1
        elif _ == 1:
            print('-This is a sample of data:')
            for i in range(4):
                print(row[i])
            _ += 1
        else:
            break
```

```
-These should be our chosen categories:
ï»¿city
state
date_time
shape
-This is a sample of data:
Chester
VA
12/12/2019 18:43
light
```

In [545]:

```
# We create some functions to handle the creation of the list, its cleansing from datas th
at have incomplete entries
# and its parsing into the Sighting class

def Create_List(File_Name, List):
    """Creates a list from a csv file, for the first 5001 lines"""
    with open(File_Name, 'r') as Raw_CSV_File:
        Read_CSV_File = csv.reader(Raw_CSV_File, delimiter=';')
        _ = 0
        for row in Read_CSV_File:
            if _ <= 5000:
                List.append([row[0],row[1],row[2],row[3]])
                _ += 1
            else:
                break

def Clean_List(List, Cleansed_List):
    """Pops out the first row of a list (the "column names" row) and purges data with Null
entries"""
    _ = 0
    for elmnt in List:
        if _ == 0:
            _ += 1
        elif all(List[_]): # all() computes an AND between all the elements of its iterabl
e argument (that must be one and only one)
            Cleansed_List.append(elmnt)
            _ += 1
        elif not all(List[_]): # we need to tell the iterator what to do in case of Null e
ntries, no break is needed because Python is lovely
            _ += 1

def Parse_List(Cleansed_List, Parsed_List):
    """Parses the Cleansed List in elements of the Sighting class, splitting date and time
through the string.split() method"""
    for elmnt in Cleansed_List:
        Sight = Sighting(elmnt[0],elmnt[1], elmnt[2].split()[0], elmnt[2].split()[1], elmn
t[3])
        Parsed_List.append(Sight)

def Unpack_List(Cleansed_List, Unpacked_List):
    """Unpacks the Sighting class"""
    for elmnt in Cleansed_List:
        Unpacked_List.append([elmnt[0],elmnt[1], elmnt[2].split()[0], elmnt[2].split()[1],
elmnt[3]])

File_Name = 'UFO_Sightings_Clean.csv'
Sightings_List = []
Cleansed_List = []
Parsed_List = []
Unpacked_List = []

Create_List(File_Name, Sightings_List)
Clean_List(Sightings_List, Cleansed_List)
Parse_List(Cleansed_List, Parsed_List)
Unpack_List(Cleansed_List, Unpacked_List)
```

```
# Let's test if the splitting and parsing has gone correctly

for i in range(100):
    print(Parsed_List[i].date)
```

```
12/12/2019
22/03/2019
17/04/2019
15/03/2009
02/04/2019
01/05/2019
10/04/2019
14/07/1973
18/06/2019
12/06/2019
11/06/2019
15/06/2018
15/08/1999
17/07/1975
17/08/2019
14/08/2019
09/08/2019
09/08/2019
06/08/2019
05/08/2019
03/08/2019
01/08/2019
29/07/2019
27/06/2019
15/06/2015
22/03/2006
11/09/2001
15/07/1979
01/06/1969
01/06/1969
27/07/1969
01/07/1970
15/07/1970
22/07/1970
01/06/1971
14/06/1971
15/07/1971
25/07/1971
12/10/1971
25/12/1971
01/07/1972
15/04/1973
15/06/1973
01/07/1973
22/09/1973
15/10/1973
15/04/1974
10/08/1974
10/08/1974
13/09/1974
18/04/1975
15/06/1975
04/07/1975
30/06/1976
01/08/1976
15/09/1976
30/10/1976
```

```
01/06/1977
01/06/1977
30/06/1977
12/07/1977
15/07/1977
01/06/1978
01/06/1978
15/08/1978
01/09/1978
20/05/1979
20/08/1979
15/09/1979
21/09/1979
03/11/1979
24/06/1980
15/08/1980
22/08/1980
14/09/1981
01/06/1982
01/06/1982
01/06/1982
01/10/1983
15/03/1984
01/06/1984
30/06/1984
15/09/1984
11/11/1984
15/11/1984
01/06/1985
01/07/1985
01/12/1985
04/04/1986
01/06/1986
13/07/1986
31/07/1986
01/08/1986
29/01/1987
14/04/1987
15/08/1987
15/08/1987
15/09/1987
08/03/1988
20/06/1988
```

Visualizing data

Now that everything is up and running, we can try some basic plotting to understand alien habits. We use `matplotlib.pyplot` , a basic plotting submodule from matplotlib.

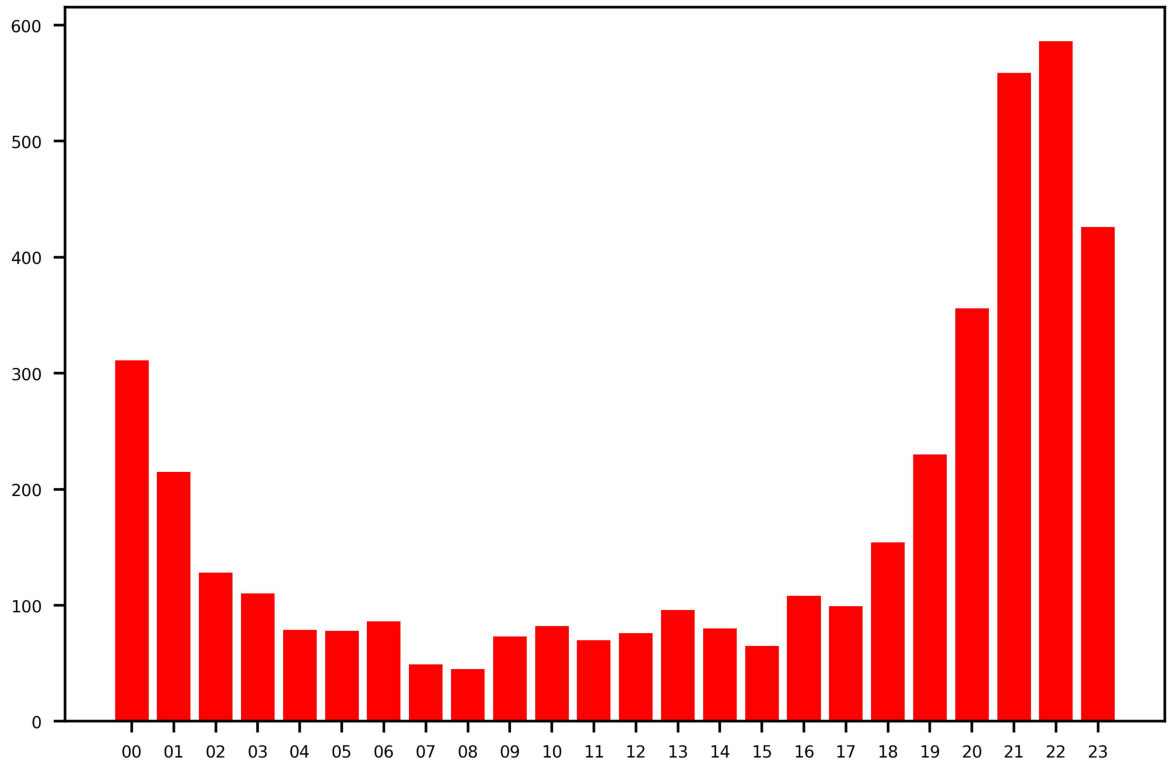
In [521]:

```
import matplotlib
from matplotlib import pyplot as plt

Data_Shapes=[]
Shapes=set()
Hystogram_Shapes = {}

for i in range(len(Parsed_List)):
    Data_Shapes.append(Parsed_List[i].shape)
    Shapes.add(Parsed_List[i].shape)

for shape in Shapes:
    Hystogram_Shapes[shape]=Data_Shapes.count(shape)
    # we use the .count method for lists to count the occurrence of 'shape' inside 'Data_S
hapes'
    # we then pass it to the Hystogram_Shapes dictionary as a value associated to the 'sha
pe' key

Data_Times=[]
Times=[*range(24)] # Notice the use of * to unpack the elements of the iterable range(), a
ctually range() is not a list object itself!
# Notice that here we initialize the list to have exactly 24 elements because we use Times
[i]="..."
# Because we are using Times[i]="...", accessing a non-existing element would throw an err
or
# The .append() method wouldn't yield an error and poses no risk of overwriting data
Keys=[]
Hystogram_Times = [*range(24)]

for i in range(len(Parsed_List)):
    Time = Parsed_List[i].time
    Hour = Time.split(':')[0] # We use the .split method on the Sighting.time string to po
p out the hour
    Data_Times.append(Hour)
for i in range(10):
    Times[i]="0{value}".format(value=i) # We are constructing a string with the possible v
alues that the .count method can encounter
    Hystogram_Times[i]=Data_Times.count(Times[i])
for i in range(10,24):
    Times[i]="{value}".format(value=i)
    Hystogram_Times[i]=Data_Times.count(Times[i])
```

In [538]:

```
for i in range(24):
    print('%s - %s' % (Times[i],Hystogram_Times[i]))
plt.bar(Times, Hystogram_Times, color='r')
plt.rcParams['figure.dpi'] = 500
plt.rcParams['font.size'] = 5.0
plt.show()
```

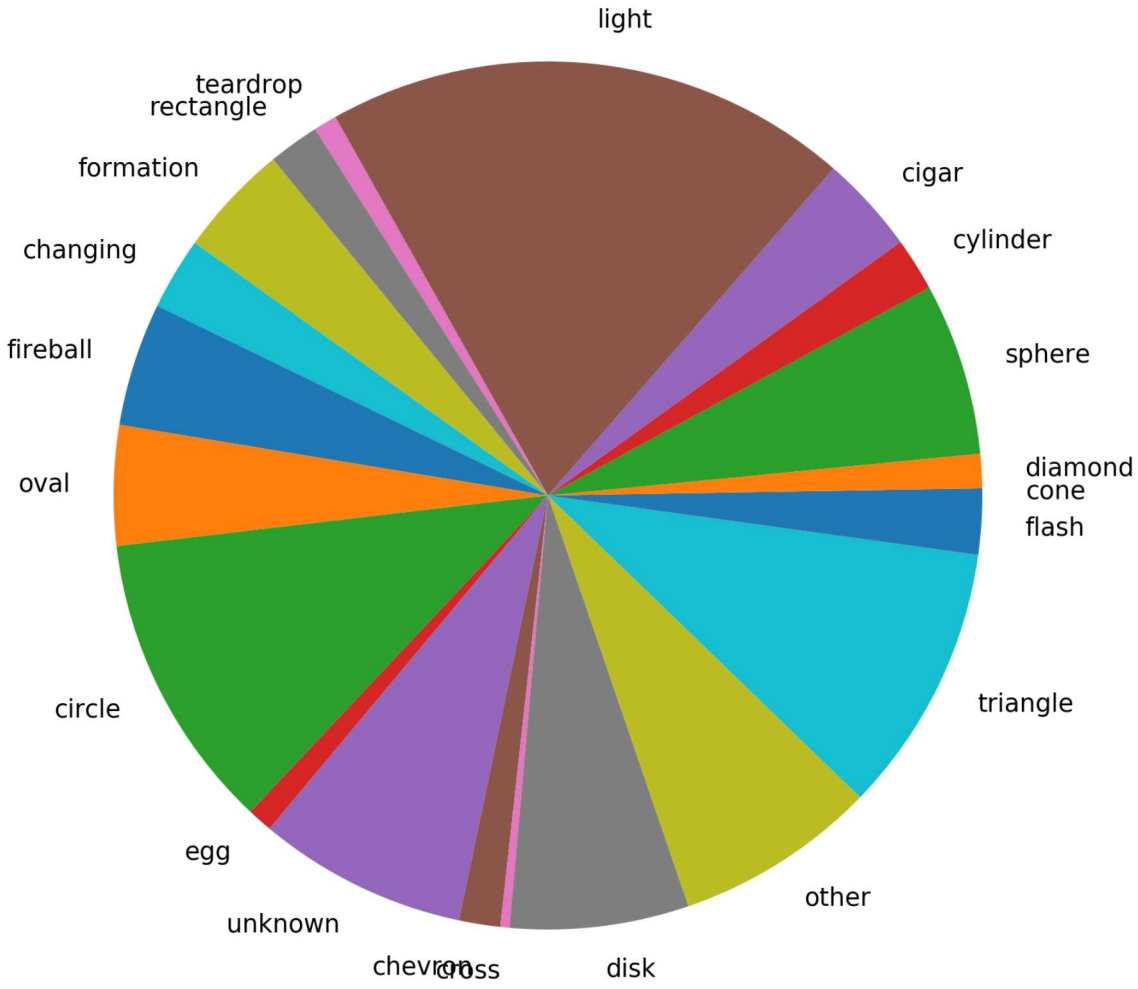
```
00 - 311
01 - 215
02 - 128
03 - 110
04 - 79
05 - 78
06 - 86
07 - 49
08 - 45
09 - 73
10 - 82
11 - 70
12 - 76
13 - 96
14 - 80
15 - 65
16 - 108
17 - 99
18 - 154
19 - 230
20 - 356
21 - 559
22 - 586
23 - 426
```



In [537]:

```
for shape in Hystogram_Shapes:
    print('%s - %s' % (shape, Hystogram_Shapes[shape]))
plt.pie( Hystogram_Shapes.values(), labels = Hystogram_Shapes.keys())
plt.rcParams['figure.dpi'] = 500
plt.rcParams['font.size'] = 5.0
plt.show()
```

```
cone - 11
diamond - 53
sphere - 267
cylinder - 82
cigar - 152
light - 814
teardrop - 36
rectangle - 79
formation - 175
changing - 112
fireball - 191
oval - 186
circle - 463
egg - 39
unknown - 324
chevron - 64
cross - 14
disk - 277
other - 313
triangle - 418
flash - 91
```



In [ ]:

```
```