

Lecture 4- Further Data Operations

Analytics Programming and Data Visualization

Lecture 4 - Further Data Operations

- ❑ Overview
- ❑ Exception Handling
- ❑ Some Useful Libraries: Pandas and NumPy
- ❑ Regular Expressions
- ❑ Text Analytics
- ❑ Summary

What is an Exception?

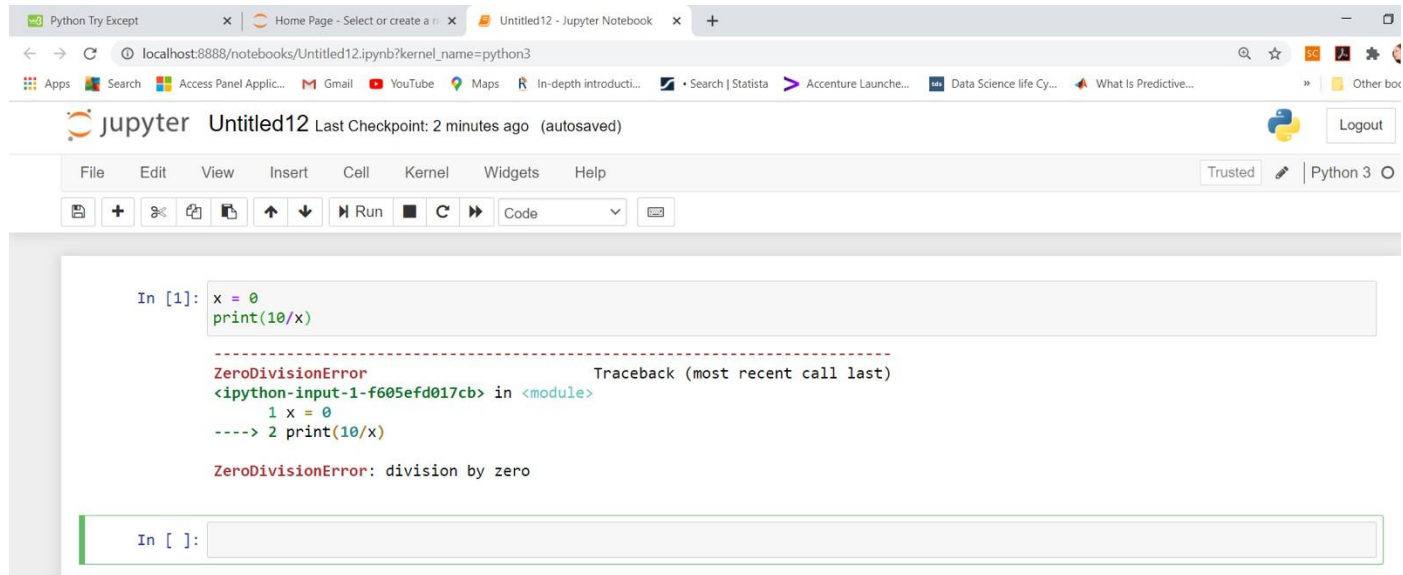
An exception is an event that occurs **during the course of program execution** that interrupts the normal flow of the program and that requires special handling.

In order for the program to run, it must be syntactically correct. If it is not, a syntax error will be flagged and the program will not run.

Exceptions are **not the same as syntax errors** as they occur while a syntactically valid program is running.



Exception Handling



The screenshot shows a Jupyter Notebook interface in a web browser. The browser's address bar displays 'localhost:8888/notebooks/Untitled12.ipynb?kernel_name=python3'. The notebook's title bar reads 'jupyter Untitled12 Last Checkpoint: 2 minutes ago (autosaved)'. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. The toolbar contains icons for file operations, running, and code execution. The main area shows a code cell with the following content:

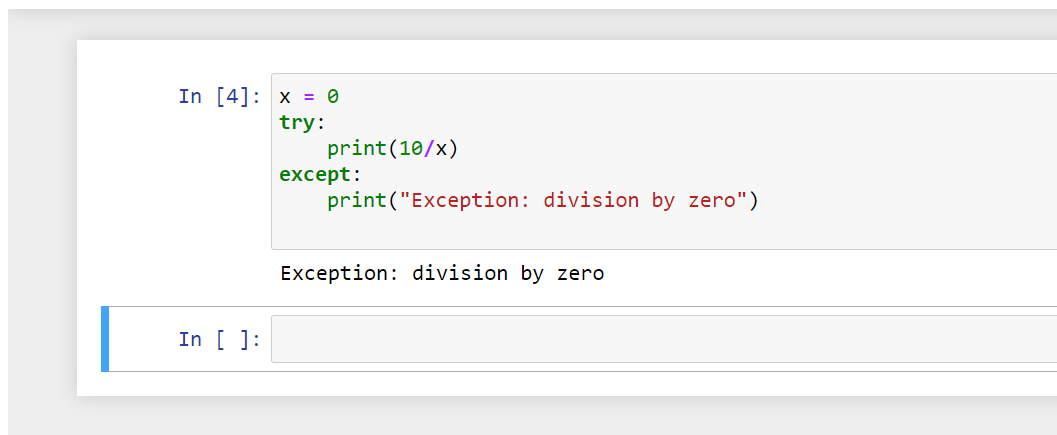
```
In [1]: x = 0
        print(10/x)
```

Below the code cell, a traceback is displayed:

```
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-1-f605efd017cb> in <module>
      1 x = 0
----> 2 print(10/x)

ZeroDivisionError: division by zero
```

At the bottom, there is an empty input prompt: 'In []: '.



The screenshot shows a Jupyter Notebook interface. The main area displays a code cell with the following content:

```
In [4]: x = 0
        try:
            print(10/x)
        except:
            print("Exception: division by zero")
```

Below the code cell, the output is shown:

```
Exception: division by zero
```

At the bottom, there is an empty input prompt: 'In []: '.

Built-in Exceptions

Python has built-in exceptions, for example:

ZeroDivisionError - thrown when an arithmetic operation involves a division by zero.

MemoryError - thrown when a program runs out of memory

OSError - thrown when an operating system error occurs, such as when a file can not be opened, read from or written to.

TypeError - thrown when an operation is applied to an object of an incorrect type.

- See <https://docs.python.org/3/library/exceptions.html> for full details of Python's built-in exceptions.

https://www.w3schools.com/python/python_ref_exceptions.asp



Try - except constructs

Try - except constructs provide a way of catching errors in a block of code and handling those errors. The basic syntax is:

try :

statement block protected by exceptional handling

except :

statement block to handle exceptions

The statements in the *try* block will be executed. If an error occurs, the block will be exited and control flow passes to the *except* block where the exception should be handled.

If no exception occurs, the *except* block will be skipped.



Try - except constructs

```
def getNumber() :  
    isnumber = False  
    while not isnumber :  
        try :  
            num = float(input('Enter a number:'))  
            isnumber = True  
        except ValueError :  
            print('You did not enter a number. Try again!')  
s = getNumber()
```

If the string entered by the user can not be converted to a *float* then the *isnumber = True* statement is never evaluated.

Instead a *ValueError* exception is thrown, try block is exited, the exception handler prints a message and the *while* block is repeated.



Try - except constructs

An *except* statement that is generic, i.e an except not followed by a particular class of exception, will trigger if any exception occurs.

A *try - except* construct can have multiple except clauses to handle different exceptions in different ways.

```
try :  
    statement block protected by exceptional handling  
except OSError:  
    statement block to handle exceptions surfaced by the OS  
except MemoryError:  
    statement block to handle memory errors
```

If an exception occurs, only the statements in a matching *except* block will be executed.



Try - except constructs

Note that using a generic except block is considered **bad programming practice** as it catches all subclasses of Exception and assumes that all will be handled in exactly the same way.

However, there are circumstances where multiple exceptions will have common exception handling code. An except statement can be followed by multiple exception classes if the code for handling for all those types of exception is identical.

```
try :  
    statement block protected by exceptional handling  
except (OSError, MemoryError) :  
    statement block to handle OS and memory errors
```



Try - except- else constructs

The *try- except* construct can also include an *else* clause containing code that will be executed only if no exception occurs. The basic syntax is:

```
try :  
    statement block protected by exceptional handling  
except :  
    statement block to handle exceptions  
else :  
    statement block executed if no exception occurs
```



Try - except- else constructs

```
try:
    filepath = './myfile.txt'
    with open(filepath, 'r') as fh : lines = fh.readlines()
except FileNotFoundError :
    print('The file {} was not found.'.format(filepath))
except IsADirectoryError :
    print('{} is a directory.'.format(filepath))
except PermissionError :
    print('Insufficient permissions to open {}'.format(filepath))
else :
    for line in lines :
        print(line)
```

If an exception is encountered, the relevant message will be printed and the else block will be skipped.

If no exception is encountered, the lines read in as part of the try block will be printed.



Try - except- finally constructs

The try - except clause can also be followed a finally clause containing code that will be executed whether an exception occurs or not. The basic syntax is:

```
try :  
    statement block protected by exceptional handling  
except :  
    statement block to handle exceptions  
finally :  
    statement block executed whether or not  
    there has been an exception
```

The code in the finally block is typically used to free resources that have been acquired during the try block.



Try - except- else example

```
try:
    filepath = './myfile.txt'
    fh = open(filepath, 'r')
    lines = fh.readlines()
except FileNotFoundError :
    print('The file {} was not found.'.format(filepath))
else :
    for line in lines : print(line)
finally:
    if 'fh' in locals: fh.close()
```

For the sake of simplicity, only one exception class is handled here. Irrespective as to whether this exception is thrown or not, the file resource will be closed.



Unhandled exceptions

An exception that is thrown during a block protected by a try statement but which is not handled by an except block will **bubble up through the call stack** until it reaches another block protected by try and except statement.

If having progressed through the call stack, no exception handler is found for the exception in question, then the program will **terminate with an error**.



Exceptions defined by modules

Modules may define their own exceptions. For instance, the module *urllib* we met in last week's material includes the *URLError* exception, defined as a subclass of *OSError*.

```
import urllib.request
try :
    response = urllib.request.urlopen('http://misspelledsite.com/')
except urllib.error.URLError as e :
    print('An error occurred when opening the URL:')
    print(str(e))
```

Note that the exception can be captured and cast to a string by adding as **<name>** after the exception class and then casting **<name>** to a string.



User-defined exceptions

There may be occasions where we need to define our own exceptions to handle events that are unique to our program. User-defined exceptions are created by creating a class based on the built-in Exception class or one of its subclasses.

```
class OddNumberError(Exception) : #
    constructor method
    def __init__(self, value) :
        self.value = str(value)
    # exception message method
    def __str__(self):
        return("An odd number was entered: " + self.value)
```


Raising an Exception

To raise an exception, the `raise` keyword is used, followed by the exception to be raised and any arguments passed to the exception.

```
try:
    i = int(input("Enter a whole even number: "))
    if i % 2 == 1 : raise OddNumberError(i)
except OddNumberError as error :
    print('An exception occurred:', error)
```



The Assert keyword

The **assert** keyword lets you test if a condition in your code returns True. If it does not, the an **AssertionError** exception will be raised.

Assertions are generally used to check one of three fundamental types of conditions:

- A **precondition** is condition that must be **true** at the **start** of a piece of code (typically, a function or a method) in order for the code to work correctly.
- A **postcondition** is a condition that should always be **true** when the piece of code finishes executing.
- An **invariant condition** is one that should always be **true** at some point during the execution of a piece of code.



Data analysis using Python

➤ Understanding the data

Dataset – Used Automobiles (CSV)

```
3,?,alfa-romero,gas,std,two,convertible,rwd,front,88.60,168.80,64.10,48.80,2548,dohc,four,130,mpfi,3.47,2.68,9.00,111,5000,21,27,13495
3,?,alfa-romero,gas,std,two,convertible,rwd,front,88.60,168.80,64.10,48.80,2548,dohc,four,130,mpfi,3.47,2.68,9.00,111,5000,21,27,16500
1,?,alfa-romero,gas,std,two,hatchback,rwd,front,94.50,171.20,65.50,52.40,2823,ohcv,six,152,mpfi,2.68,3.47,9.00,154,5000,19,26,16500
2,164,audi,gas,std,four,sedan,fwd,front,99.80,176.60,66.20,54.30,2337,ohc,four,109,mpfi,3.19,3.40,8.00,102,5500,24,30,13950
2,164,audi,gas,std,four,sedan,4wd,front,99.40,176.60,66.40,54.30,2824,ohc,five,136,mpfi,3.19,3.40,8.00,115,5500,18,22,17450
2,?,audi,gas,std,two,sedan,fwd,front,99.80,177.30,66.30,53.10,2507,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,15250
1,158,audi,gas,std,four,sedan,fwd,front,105.80,192.70,71.40,55.70,2844,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,17710
1,?,audi,gas,std,four,wagon,fwd,front,105.80,192.70,71.40,55.70,2954,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,18920
1,158,audi,gas,turbo,four,sedan,fwd,front,105.80,192.70,71.40,55.90,3086,ohc,five,131,mpfi,3.13,3.40,8.30,140,5500,17,20,23875
0,?,audi,gas,turbo,two,hatchback,4wd,front,99.50,178.20,67.90,52.00,3053,ohc,five,131,mpfi,3.13,3.40,7.00,160,5500,16,22,?
2,192,bmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2395,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16430
0,192,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2395,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16925
0,188,bmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2710,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,21,28,20970
0,188,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2765,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,21,28,21105
1,?,bmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3055,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,20,25,24565
0,?,bmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3230,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,16,22,30760
0,?,bmw,gas,std,two,sedan,rwd,front,103.50,193.80,67.90,53.70,3380,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,16,22,41315
0,?,bmw,gas,std,four,sedan,rwd,front,110.00,197.00,70.90,56.30,3505,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,15,20,36880
2,121,chevrolet,gas,std,two,hatchback,fwd,front,88.40,141.10,60.30,53.20,1488,1,three,61,2bbl,2.91,3.03,9.50,48,5100,47,53,5151
```

<https://archive.ics.uci.edu/ml/machine-learning-databases/autos/>

Each of the attributes in the dataset

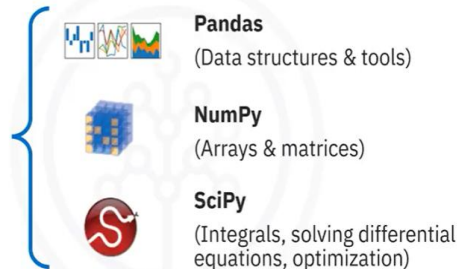
No.	Attribute name	attribute range	No.	Attribute name	attribute range
1	symboling	-3, -2, -1, 0, 1, 2, 3.	14	curb-weight	continuous from 1488 to 4066.
2	normalized-losses	continuous from 65 to 256.	15	engine-type	dohc, dohcv, l, ohc, ohcf, ohcv, rotor.
3	make	audi, bmw, etc.	16	num-of-cylinders	eight, five, four, six, three, twelve, two.
4	fuel-type	diesel, gas.	17	engine-size	continuous from 61 to 326.
5	aspiration	std, turbo.	18	fuel-system	1bbl, 2bbl, 4bbl, idi, mfi, mpfi, spdi, spfi.
6	num-of-doors	four, two.	19	bore	continuous from 2.54 to 3.94.
7	body-style	hardtop, wagon, etc.	20	stroke	continuous from 2.07 to 4.17.
8	drive-wheels	4wd, fwd, rwd.	21	compression-ratio	continuous from 7 to 23.
9	engine-location	front, rear.	22	horsepower	continuous from 48 to 288.
10	wheel-base	continuous from 86.6 to 120.9.	23	peak-rpm	continuous from 4150 to 6600.
11	length	continuous from 141.1 to 208.1.	24	city-mpg	continuous from 13 to 49.
12	width	continuous from 60.3 to 72.3.	25	highway-mpg	continuous from 16 to 54.
13	height	continuous from 47.8 to 59.8.	26	price	continuous from 5118 to 45400.

Target (Label)

Python libraries

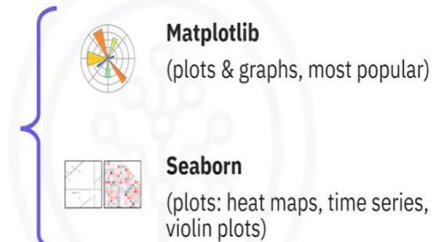
Scientifics computing libraries in Python

1. Scientifics Computing Libraries



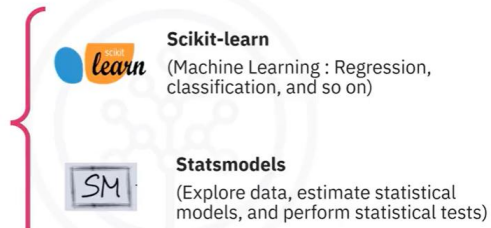
Visualization libraries in Python

2. Visualization Libraries



Algorithmic libraries in Python

3. Algorithmic Libraries



Importing and Exporting data

Importing data: Process of loading and reading data from various resources

- Data formats: .csv, .json, .xlsx,
- File path of dataset: where data is stored
e.g., Computer: /Downloads/mydata.csv
Internet: <https://archive.ics.uci.edu/autos/imports-85.data>



Importing .csv data using Panda library

```
import pandas as pd
```

```
url = "https://archive.ics.uci.edu/ml/machine-learningdatabases/autos/imports-85.data"
```

```
df = pd.read_csv(url)
```

```
import pandas as pd
```

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data"
```

```
df = pd.read_csv(url, header = None)
```



Printing the data in Pandas

Without header

- **df** prints the entire dataframe (not recommended for large datasets)
- **df.head(n)** to show the first n rows of data frame
- **df.tail(n)** shows the bottom n rows of data frame

df.head()

n=5 {

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21	22	23	24	25
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27	13495
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27	16500
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19	26	16500
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24	30	13950
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18	22	17450

Printing the data in Pandas

With header

- Replace default header (by `df.columns = headers`)

```
headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "num-of-doors","body-style",  
           "drive-wheels","engine-location","wheel-base", "length","width","height","curb-weight","engine-type",  
           "num-of-cylinders", "engine-size","fuel-system","bore","stroke","compression-ratio","horsepower", "peak-rpm","city-mpg","highway-mpg","price"]
```

```
df.columns=headers
```

```
df.head(5)
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	130	mpfi	3.47	2.68	9.0	113
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	130	mpfi	3.47	2.68	9.0	113
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	152	mpfi	2.68	3.47	9.0	113
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	109	mpfi	3.19	3.40	10.0	113
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	136	mpfi	3.19	3.40	8.0	113

Exporting Pandas dataframe

Path= "C:\Windows\.....\automobile.csv"

df.to_csv(path)

Data Format	Read	Save
csv	pd.read_csv()	df.to_csv()
json	pd.read_json()	df.to_json()
Excel	pd.read_excel()	df.to_excel()
sql	pd.read_sql()	df.to_sql()



Data type and distribution in Pandas

Basic Insights of Dataset - Data Types

Pandas Type	Native Python Type	Description
object	string	numbers and strings
int64	int	Numeric characters
float64	float	Numeric characters with decimals
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	time data.

To check datatype in Pandas: `dataframe.dtypes`
df.dtypes

To return statistical summary of data: `dataframe.describe()`
df.describe(), df.info()

	symboling	wheel-base	length	width	height	curb-weight	engine-size	compression-ratio	city-mpg	highway-mpg
count	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000
mean	0.834146	98.756585	174.049268	65.907805	53.724878	2555.565854	126.907317	10.142537	25.219512	30.751220
std	1.245307	6.021776	12.337289	2.145204	2.443522	520.680204	41.642693	3.972040	6.542142	6.886443
min	-2.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	7.000000	13.000000	16.000000
25%	0.000000	94.500000	166.300000	64.100000	52.000000	2145.000000	97.000000	8.600000	19.000000	25.000000
50%	1.000000	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	9.000000	24.000000	30.000000
75%	2.000000	102.400000	183.100000	66.900000	55.500000	2935.000000	141.000000	9.400000	30.000000	34.000000
max	3.000000	120.900000	208.100000	72.300000	59.800000	4066.000000	326.000000	23.000000	49.000000	54.000000

```
symboling          int64
normalized-losses  object
make              object
fuel-type         object
aspiration        object
num-of-doors      object
body-style        object
drive-wheels      object
engine-location   object
wheel-base       float64
length           float64
width            float64
height           float64
curb-weight       int64
engine-type       object
num-of-cylinders  object
engine-size       int64
fuel-system       object
bore              object
stroke            object
compression-ratio float64
```

Dealing with missing values

- when no data value is stored for feature for a particular observation then we say a missing value.
- And this is represented a '?', N/A, zero or just a blank cell .

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front

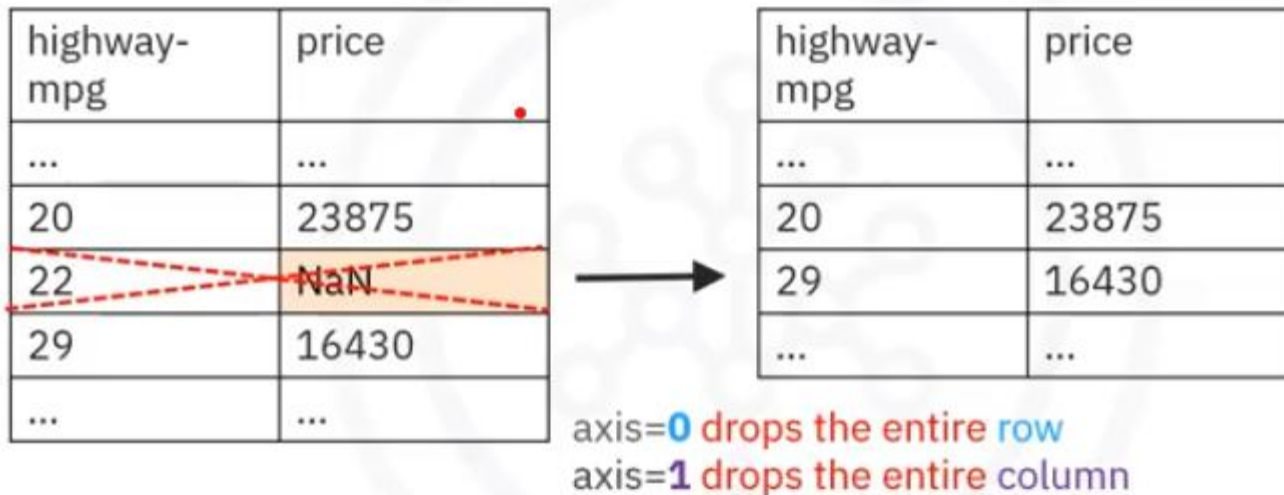
How to deal with missing data?

- ❖ Check with data collection source
- ❖ Drop the missing value
 - Drop the whole variable
 - drop the data entry
- ❖ Replace the missing values
 - replace it with an average (of similar datapoints)
 - replace it by frequency
 - replace it based on other functions
- ❖ Leave it as missing data



How to drop missing values in Python

- Use `dataframes.dropna()`:



highway-mpg	price
...	...
20	23875
22	NaN
29	16430
...	...

axis=0 drops the entire row
axis=1 drops the entire column

```
df.dropna(subset=["price"], axis=0, inplace = True)
```

How to replace missing values in Python

Use `dataframe.replace(missing_value, new_value)`:

normalized-losses	make
...	...
164	audi
164	audi
NaN	audi



normalized-losses	make
...	...
164	audi
164	audi
162	audi
158	audi
...	...

```
mean = df["normalized-losses"].mean()
```

```
df["normalized-losses"].replace(np.nan, mean)
```



❖ **Check with data collection source**

❖ Drop the missing value

Drop the whole variable

drop the data entry

❖ Replace the missing values

replace it with an average (of similar datapoints)

replace it by frequency

replace it based on other functions

❖ **Leave it as missing data**



Useful NumPy Operations

Some Useful Libraries: NumPy

NumPy

Numerical Python

- Linear Algebra
- Fourier Transformations
- Random Numbers

Can operate directly on arrays and matrices

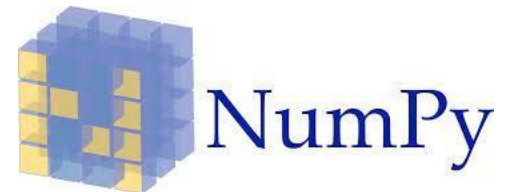
Algorithms implemented with performance in mind

Extensive set of numerical types

Arrays stored more efficiently than base Python lists

Files memory-mapped to arrays

Large portions written in C



Some Useful Libraries: NumPy

NumPy Arrays

NumPy has a multi-dimensional array object called `ndarray`

An `ndarray` array object contains

- The actual data
- Some metadata describing the data

Generally used with homogeneous data

Zero-based indexing



Some Useful Libraries: NumPy

NumPy Arrays

```
In [50]: import numpy as np

myArr = np.array([1,2,3,4,5,6,7,8,9,10])

print(myArr)
print(len(myArr))
print(type(myArr))
print(np.__version__)

[ 1  2  3  4  5  6  7  8  9 10]
10
<class 'numpy.ndarray'>
1.18.5
```



Some Useful Libraries: NumPy

0-D arrays (Scalars): elements in an array - each value in an array is a 0-D array. E.g 42

1-D array (uni-dimensional): array that has 0-D arrays as its elements. E.g. [1,2,3]

2-D array (matrix or 2nd order tensors): array that has 1-D arrays as its elements. E.g. [[1,2,3],[4,5,6]]

3-D array (3rd order tensor): array that has 2-D arrays (matrices) as its elements. E.g. [[[1,2,3],[4,5,6]],[7,8,9],[10,11,12]]

N-D array: an array can have any number of dimensions can be created by using the **ndmin** argument as the 2nd argument in the array method of numpy package



Some Useful Libraries: NumPy

NumPy Arrays

```
In [57]: import numpy as np

arr0d = np.array(42)
arr1d = np.array([1,2,3])
arr2d = np.array([[1,2,3],[4,5,6]])
arr3d = np.array([[[1,2,3],[4,5,6]], [[7,8,9],[10,11,12]]])
arr6d = np.array([[[[1,2,3],[4,5,6]], [[7,8,9],[10,11,12]]], ndmin=6)

print(arr0d)
print(arr1d)
print(arr2d)
print(arr3d)
print(arr6d)
print("\n")
print(arr6d.shape)
print(arr3d[0,0,0])

42
[1 2 3]
[[1 2 3]
 [4 5 6]]
[[[ 1  2  3]
 [ 4  5  6]]

 [[ 7  8  9]
 [10 11 12]]]
[[[[[ 1  2  3]
 [ 4  5  6]]

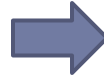
 [[ 7  8  9]
 [10 11 12]]]]]]

(1, 1, 1, 2, 2, 3)
1
```

Some Useful Libraries: NumPy

The `array` function creates an array from an array-like object (e.g., a Python list). In this example, a list of arrays was passed to the `array` function.

```
print(m.dtype)
print(m.dtype.itemsize)
```



```
int64
8
```

The datatype of the array is `int64`. The size in bytes for the `int64` is 8 .

NumPy Datatypes

Below is a list of all data types in NumPy and the characters used to represent them.

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type (void)

Some Useful Libraries: NumPy

NumPy Arrays Examples

```
myType = numpy.dtype([('name', numpy.str_, 40),  
                      ('numitems', numpy.int32),  
                      ('price', numpy.float32)])  
print(myType)  
  
myItems = numpy.array([('Item 1', 13, 4.20),  
                      ('Second Item', 2, 101.99),  
                      ('Another Item', 1, 24.99)], dtype=myType)  
print(myItems[1])
```



```
[('name', 'S40'), ('numitems', '<i4'), ('price', '<f4')]  
('Second Item', 2, 101.98999786376953)
```



Some Useful Libraries: NumPy

NumPy Array Methods

NumPy arrays support a wide range of methods for manipulation of data

- ☐ Slicing
- ☐ Stacking
- ☐ Reshaping
- ☐ Splitting
- ☐ Conversion to other datatypes
- ☐ Etc.

We will do some exercises with these methods during this week's lab session.



ARRAY MANIPULATION

Slicing an ndarray is very similar to slicing a regular array, except that we now have to specify slicing indices **for each dimension**:

```
arr2d = np.array([[3, 4, 5], [6, 7, 8], [9, 10, 11]])  
print(arr2d[1:3, 2])  
  
> [ 8 11]
```

Slicing in two dimensions i (rows) and j (columns) takes the form:

```
arrayName[i, j]
```

where i and j can be single indices or ranges.

Ranges are specified as *startindex:stopindex* and return a result inclusive of the *startindex* but exclusive of the *stopindex*.



ARRAY MANIPULATION

Stacking is used to join a sequence of arrays to form a **new axis**.

```
import numpy as np
a1 = np.array([1,2,3])
a2 = np.array([4,5,6])
print(np.stack((a1,a2),axis = 0))

[[1 2 3]
 [4 5 6]]
print(np.stack((a1,a2),axis = 1))

[[1 4]
 [2 5]
 [3 6]]
```



Some Useful Libraries: NumPy

NumPy Common Functions

NumPy also supplies a large suite of functions for other data manipulation tasks

- ☐ Loading data files
- ☐ Calculating statistics
- ☐ Performing functional transformations
- ☐ Sampling distributions
- ☐ Fitting Polynomials
- ☐ Smoothing data
- ☐ Calculating derivatives
- ☐ Trigonometric functions
- ☐ Solving linear systems

We will do some exercises with (some of) these methods during this week's lab session.



Regular Expressions

Regular Expressions

- ▮ Regular Expressions
- ▮ Used as a compact way to describe complex patterns in text
- ▮ Can be used to search for patterns in text and subsequently modify the patterns in text
- ▮ Can be used to launch programmatic actions that depend on

‘Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.’

Jamie Zawinski <alt.religion.emacs> (08/12/1997)



Regular Expressions

▮ Regular Expressions

▮ Used extensively in other tools

- ▮ bash command-line
 - ▮ grep
 - ▮ sed
 - ▮ awk
- ▮ Perl
- ▮ Other programming languages
 - ▮ R
 - ▮ C
 - ▮ Java
 - ▮ .NET languages



Regular Expression Functions

- **findall** - returns a list containing all matches.

```
import re
re.findall('\d', '1 and 2 and 3')
['1', '2', '3']
```

- **search** - returns the first match as a Match object.

```
import re
Match = re.search('\d', 'And 1 and 2 and 3')
print(match)
<re.Match object; span=(4, 5), match='1'>
```

□ Note: this example uses the special sequence `\d`, which will match any digit character.



Regular Expression Functions

- **split** - returns a list with the string split at each match

```
import re
split = re.split('\s', 'Regular\textexpressions are\npowerful')
print(split)

['Regular', 'expressions', 'are', 'powerful']
```

Note: this example uses the special sequence `\s`, which will match any whitespace character

- **sub** - replaces one or more matches with a new string

```
import re
print(re.sub('or', 'and', 'A or B or C or D'))

>A and B and C and D
```



Metacharacters

Metacharacters are characters in an expression that have a special meaning. [] square brackets are used to enclose a set of characters:

```
import re
find = re.findall('[1,4,9]', '1234568')
print(find)
['1', '4']

import re
find = re.findall('[a-c]', 'eafbgc')
print(find)
['a', 'b', 'c']
```



Metacharacters

. matches any **single character**.

^ matches the **start of a string**:

```
import re
find = re.findall('^...', 'The quick brown fox')
print(find)
['Th']
```

\$ matches the **end of a string**:

```
import re
find = re.findall('again$', 'Again and again and again')
print(find)
['again']
```



Metacharacters

- * matches **zero or more occurrences of a string or special character.**

```
import re
find = re.findall('gan*', 'Again and again and again')
print(find)
['ga', 'ga', 'ga']
```

- + matches **one or more occurrences of a string or special character.**

```
import re
find = re.findall('gai+', 'Again and again and again')
print(find)
['gai', 'gai', 'gai']
```

Metacharacters

{ matches the **exact number of occurrences of a string or special character.**

```
import re
find = re.findall('[l,o]{2}','We always watch football')
print(find)
['oo','ll']
```

| matches **either one or both of the strings or special characters on either side.**

```
import re
s = 'We always watch football or cricket'
find = re.findall('always|football|cricket',s)
print(find)
['always','football','cricket']
```



Special Sequences

- Regular expressions provide a variety of special sequences that permit matching against specific parts of a string or particular character sequences. We have already met some of these in previous slides. Further examples are shown below:
- **\A** - matches if the specified characters occur at the start of a string:

```
import re
find = re.findall('\AThe', 'The quick brown fox. The lazy dog')
print(find)
['The']
```

- **\D** - matches any character that is not a decimal digit:

```
import re
find = re.findall('\D', '1 and 2 or 3')
print(find)
>[' ', 'a', 'n', 'd', ' ', ' ', 'o', 'r', ' ']
```

Special Sequences

- `\w` - Matches any **word characters** (alphanumeric characters) or the underscore:

```
import re
s = '!AabcC&_'
find = re.findall(r'\w',s)
print(find)
['A', 'a', 'b', 'c', 'C', '_']
```

- `\b` - matches a **string or special character** when it occurs at a word boundary:

```
import re
s = 'We always watch football or cricket'
find = re.findall(r'\bal',s)
print(find)
['al']
```

Capture Groups

- Capture groups are denoted by parentheses and are used to **group the regex** enclosed within them. They capture matches for the regex into numbered groups that can be **extracted** or used with a **numbered back-reference**.
- The example below checks for the presence of an IP address and if found, extracts each part of the dotted quad.

```
import re
ipregex = r'\b([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\b'
string = 'The IP address 192.168.1.1 is a class C address.'
match = re.findall(ipregex, string)
print(match)
[('192', '168', '1', '1')]
```



Regular Expression Flags

- Most regular expression functions in the `re` module can take an optional flags argument. Flags allow us to change the way the parser interprets the regular expressions. There are numerous regular expression flags, including:
 - **`re.M` or `re.MULTILINE`** - The start and end of string anchors will also match newlines
 - **`re.S` or `re.DOTALL`** - The dot metacharacter will also match a newline.
 - **`re.I` or `re.IGNORECASE`** - Matches on alphabetic characters will be case-insensitive.
- An example using `re.IGNORECASE` is shown on the next slide.



Regular Expression Flags

```
import re
s = 'The quick brown fox jumped over the lazy dog'
find = re.findall('the', s)
print(find)
['the']

find = re.findall('the', s, re.IGNORECASE)
print(find)
['The', 'the']
```



Regular Expression Examples

Regular Expressions

▮ The Python re Module

```
import re

def re_show(pat, s):
    print(re.compile(pat, re.M).sub("{\g<0>}", s.rstrip()))
    print("\n")

s = """Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go."""

re_show("a", s)
```

```
M{a}ry h{a}d {a} little l{a}mb.
And everywhere th{a}t M{a}ry
went, the l{a}mb w{a}s sure
to go.
```

See

<https://docs.python.org/3.7/library/re.html?highlight=regular%20expressions>

Regular Expressions

▮ The Python re Module

```
import re

def re_show(pat, s):
    print(re.compile(pat, re.M).sub("{\g<0>}", s.rstrip()))
    print("\n")

s = """Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go."""

re_show("Mary", s)
```

```
{Mary} had a little lamb.
And everywhere that {Mary}
went, the lamb was sure
to go.
```

Regular Expressions

▮ The Python re Module

Examples

```
s = """Special characters must be escaped.*"""
re_show(r".*", s)
re_show(r"\.\*", s)
re_show("\\\\", r"Python \ escaped \ pattern")
re_show(r"\\", r"Regex \ escaped \ pattern")
```

```
{Special characters must be escaped.*}

Special characters must be escaped{.*}

Python {\} escaped {\} pattern

Regex {\} escaped {\} pattern
```

It is usually easier to compose regular expression strings by quoting them as 'raw strings' with an initial 'r'.

Regular Expressions

▮ The Python re Module

```
s = """Mary had a little lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go."""
```

```
re_show(r"^Mary", s)  
re_show(r"Mary$", s)
```

“^” matches the beginning of a line

“\$” matches the end of a line

```
{Mary} had a little lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go.
```

```
Mary had a little lamb.  
And everywhere that {Mary}  
went, the lamb was sure  
to go.
```

Regular Expressions

▮ The Python re Module

```
s = """Mary had a little lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go."""
```

```
re_show(r".a", s)
```

‘.’ matches any character

```
{Ma}ry {ha}d{ a} little {la}mb.  
And everywhere t{ha}t {Ma}ry  
went, the {la}mb {wa}s sure  
to go.
```



Regular Expressions

▮ The Python re Module

```
s = """Mary had a little lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go."""
```

```
re_show(r"[a-z]a", s)  
re_show(r"[a]h[a]", s)
```

```
Mary {ha}d a little {la}mb.  
And everywhere t{ha}t Mary  
went, the {la}mb {wa}s sure  
to go.
```

```
Mary {ha}d a little {la}mb.  
And everywhere t{ha}t Mary  
went, the {la}mb was sure  
to go.
```

A set of characters can be matched by placing the characters in square brackets

Regular Expressions

▮ The Python re Module

```
s = """Mary had a little lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go."""  
  
re_show(r"lamb|Mary", s)
```

```
{Mary} had a little {lamb}.  
And everywhere that {Mary}  
went, the {lamb} was sure  
to go.
```

The alternation operator, '|', specifies that either of set of two whole expressions should be matched.

This can be extended to more than two subexpressions, e.g.:

“lamb|Mary|little”

Regular Expressions

▮ The Python re Module

Example

```
s = """Mary had a little lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go."""
```

```
re_show(r"t+", s)  
re_show(r"y?w", s)
```

```
Mary had a li{tt}le lamb.  
And everywhere {t}ha{t} Mary  
wen{t}, {t}he lamb was sure  
{t}o go.
```

```
Mary had a little lamb.  
And ever{yw}here that Mary  
{w}ent, the lamb {w}as sure  
to go.
```

Quantifiers

‘+’ specifies that an expression is matched one or more times.

‘?’ specifies that an expression is matched zero or more times.

‘*’ specifies that an expression is matched zero or more times, as many repetitions as are possible.

Regular Expressions

▮ The Python `re` Module

Examples

`*?`, `+?`, `??`

The `'*'`, `'+'`, and `'?'` qualifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE `<.*>` is matched against `'<a> b <c>'`, it will match the entire string, and not just `'<a>'`. Adding `?` after the qualifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using the RE `<.*?>` will match only `'<a>'`.

`(?=...)`

Matches if `...` matches next, but doesn't consume any of the string. This is called a *lookahead assertion*. For example, `Isaac (?=Asimov)` will match `'Isaac '` only if it's followed by `'Asimov'`.

Regular Expressions

▮ The Python re Module Examples

```
s = """Mary had a little lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go."""  
  
re_show(r"t{2}", s)
```

```
Mary had a li{tt}le lamb.  
And everywhere that Mary  
went, the lamb was sure  
to go.
```

Counts

Curly braces can determine a precise count of the number of matches to make

Min and Max counts can also be specified, e.g.:

{2,5} or { ,8}



Regular Expressions

▮ The Python re Module Examples

We can use regular expression techniques to replace text that matches a certain pattern with some substitute text.

```
def re_new(pat, rep, s):  
    print(re.sub(pat, "{"+rep+"}", s))  
    print("\n")  
  
s = "The zoo had wild dogs, bobcats, lions, and other wild cats."  
  
re_new("cat", "dog", s)
```

```
The zoo had wild dogs, bob{dog}s, lions, and other wild {dog}s.
```





Text Analytics

Will be covered through lab exercises

Text Analytics

▮ Text Analytics Overview

▮ Natural Language Processing (NLP)

- Artificial Intelligence
- Generate and *understand* natural languages

▮ Information Retrieval

- Retrieve relevant information from within a corpus of documents
- Think Web Search Engines / Library Systems



Text Analytics

▮ Natural Language Processing (NLP)

▮ Applications

- ▣ Sentiment Analysis
- ▣ Topic Modelling
- ▣ Relationship Extraction
- ▣ Automatic Summarization
- ▣ Language Detection
- ▣ Chatbots
- ▣ Speech Engines
- ▣ Security
 - ▣ Identification of entities masquerading as other entities
 - ▣ Identification of non-human generated content



Text Analytics

▮ Natural Language Processing (NLP)

▮ The Natural Language Toolkit (NLTK)

- ▣ Python library
- ▣ Extensive set of function for processing and pre-processing text and natural language content
 - ▣ Word / Sentence Tokenisation
 - ▣ Synonyms /Antonyms
 - ▣ Stemming
 - ▣ Lemmatisation
 - ▣ Stop-word removal
 - ▣ Speech Tagging
 - ▣ Chunking/Chinking
 - ▣ Named Entity Recognition
 - ▣ Anaphora Resolution
 - ▣ Word Sense Disambiguation
 - ▣ Machine Learning Classification

We will do some exercises with (some of) these methods during this week's lab session.

The NLTK can be used in conjunction with the **scikit-learn** Python library for powerful text based analytics and machine learning.

See also https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

Summary

- ▮ Overview
 - ▮ Exception Handling
 - ▮ Some Useful Libraries: NumPy
 - ▮ Regular Expressions
 - ▮ Text Analytics
-
- ▮ Bibliography / References
 - ▮ Mertz, D. Text Processing in Python. Addison-Wesley.
 - ▮ Idris, I. (2013). NumPy Beginner's Guide. Packt Publishing
 - ▮ <https://docs.python.org>
 - ▮ <https://scikit-learn.org>

