1. Python Basics
2. Numpy, Pandas
3. Web Scraping
4. Sup, Unsup
5. Tensorflow

Introduction to Python for Data Science

Dr Ekaterina Abramova

Lecture 1, AM02

Autumn 2021

Overview of Spyder + Setup

- Download all files from Session1 on Canvas
- Put all `.py` files and `.csv` data files into your folder `Documents -> AM02 -> Code`
- Open the environment you created, `spyder2021`, in Spyder
- From Spyder open file `L1_Code.py` (it contains an introductory example and all of the code from the lectures slides).

Magic Commands

Magic commands are

- IPython's special commands, which are not built into Python.
- *Command-line programs* that are run inside IPython system (i.e. Console).
- Any command which begins with `%` is a *line magic* command.

Useful commands:

`%quickref`	Quick Reference Card to all magic commands
`%conda --version`	Conda's current version
`%reset`	Delete all object from the current session
`%clear`	Delete history from IPython Console
`%time`	Execution time of a single statement
`%timeit`	Average of execution times of a single statement

```
In [8]: %quickref
%cd:
    Change the current working directory.
%clear:
    Clear the terminal.
```

(Useful for timing execution of `for` loops vs numpy arrays: to be discussed later).

Allows `conda` / `conda-forge` / `pip` commands inside IPython! e.g. install, update etc. Will only affect the environment currently opened in Spyder (i.e. `spyder2021`).

```
In [1]: %conda install numpy
```

```
In [1]: %pip install tweepy
```

Library Management Systems (1)

There are three ways to install packages, using:

1. **[1ST CHOICE]** **`conda`** : Anaconda's **official** Python package repository (1500 packages available)
   - Install packages with **`conda install package`**
   - Keep packages up-to-date with **`conda update --all`**

For example, to install a library `matplotlib` into our environment `spyder2021`, you have several options:

A. Terminal (Mac) / Anaconda Prompt (Windows)

a) Working from (base):

```
(base) NAME BookPro:~ NAME $ conda install -n spyder2021 matplotlib
```

b) Working from (`spyder2021` environment):

```
(latest-spyder) NAME BookPro:~ NAME $ conda install matplotlib
```

B. Directly from Spyder's IPython Console!

```
Console 1/A
Python 3.9.2 | packaged by conda-forge | (default, Feb 21 2021
Type "copyright", "credits" or "license" for more information.

IPython 7.22.0 -- An enhanced Interactive Python.
In [1]: %conda install matplotlib
```

Reminder: all that follows the % sign is read as a command-line instruction inside the IPython Console (otherwise it only recognizes Python commands!).

Likewise, use % to keep all packages and their dependencies up to date:

```
In [2]: %conda update --all
Collecting package metadata
Solving environment: done
```

Easiest option is to work with magic commands from Spyder's Console.

Library Management Systems (2)

If package is not available under conda, next choice is:

2. **[2nd CHOICE]** **`conda-forge`** : additional **community** repository with conda packages (link)
   ✓ Install packages with: **`conda install -c conda-forge package`**

Search if a package of your interest is available on the community's website:

ANACONDA.ORG

Again, we will want to install the package only to our environment `spyder2021`. Using the % (magic command) from Spyder's Console of our opened environment:

```
Python 3.9.2 | packaged by conda-forge | (default, Feb 21 2021,
Type "copyright", "credits" or "license" for more information.

IPython 7.22.0 -- An enhanced Interactive Python.
In [1]: %conda install -c conda-forge packageName
```

***Advanced Note**: It is recommended that when you eventually need to install packages from conda-forge, you should create another environment, containing conda-forge packages only.*

Library Management Systems (3)

The last option is using Python Package Index (PyPI). We will try to stick with conda where possible, since pip manages packages but conda manages both packages and environments.

3. **[3rd CHOICE]** `pip` : (link) (nearly 300,000 packages available!)
   - ✓ Install packages with: `pip install package`

   Search if a package is available on website: https://pypi.org

Advanced Note: *Spyder developers strongly recommend not to mix Conda and Pip! Therefore, we are advised to create another environment, into which we install pip packages only.*

Once a new environment is ready, we would use the same approach of installing packages using magic command from Spyder Console.

Functions – **Built-in**

Function – a named group of statements used to perform a specific task:

- Decomposition / **modularity** (creates structure)
- Readability, abstraction (hiding detail), code **re-usability**
- A command ending in brackets `()` is typically a function e.g. `print()` `len()` `type()`

Function invocation ("**function call**") – performed by typing out function name and passing relevant arguments inside the brackets, `function_name(argument)` e.g.:

- `round(6.7)  #` 6.7 is termed the 'argument'
- when argument is an expression, it is evaluated first, then passed into the function e.g. `round(14.7 – 4)`

Value of invocation ("**returned value**") – output from the function (can be assigned to a variable, e.g.:

- return value is 7 i.e. it can be assigned to a variable: `x = round(6.7)`

Loading a Library

There are 3 ways to load components of a library using the `import` statement (`PACKAGE` – library name; `NAME` – name of the needed item e.g. function or variable).

✅
- Import the whole library. To access an item of this library after loading, use the dot notation.
  `import library` (syntax: `library.function`) – use if `library` name is short.

✅
- Import the whole library and give it a shorthand name of your choice. This makes it quicker to type out.
  `import library as lb` (syntax: `lb.function`) – use if `library` name is long.

✅
- Import only the necessary function (or variable) from the library.
  `from library import function` (syntax: `function`) – use if only need a 1 item from the library.
  `from library import function1, function2, function3` – use if need several items.

The following way of loading a library is for the advanced user:

X
- Load all the items from the library - only use when 100% sure the entire contents are necessary.
  `from library import *` (direct use of **all** functions) – bad practice known as 'wildcard import'.

Library Loading Examples

✅

```
#1. import PACKAGE -> load a whole package (good practice)
import numpy                   # load numpy package
y1 = numpy.array([1, 2, 3]) # create an array
```

✅

```
#2. import PACKAGE as SHORTHANDNAME -> load a whole package (good practice)
import numpy as np             # load numpy module and give it a shortered name
y2 = np.array([1, 2, 3])       # create an array
```

✅

```
#3. from PACKAGE import NAME -> statement specific import (good practice)
from numpy import array        # load a particular function from numpy package
y3 = array([1, 2, 3])          # create an array
```

Can't use np.array() or numpy.array()

X

```
#4. from PACKAGE import * -> import all items (BAD practice)
from numpy import *
z1 = round(6.7)     # rounds float to 7
# here Python uses base distribution's round() function which results in an int
# i.e. we may not get the intended answer (e.g. wanted float but got int!)
# If we actually wanted to avoid the clash, do not do wild card imports, i.e.
z2 = np.round(6.7) # answer is 7.0 float
```

Library Listing Resources

- E.g. load `math` library and list its directory (check resources)

```
In [1]: import math

In [2]: dir(math)
Out[2]:
['__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

Lists what is contained inside the library. To use any of those items need the dot notation: `math.item`

Note: both variables (pi) and functions (`exp`, `sqrt`) are contained inside the library directory.

- Access from resources item: number `pi`

```
In [3]: math.pi
Out[3]: 3.141592653589793
```

`float` - Scientific Notation

- Scientific notation – is used to represent very large/small numbers.

Decimal Notation	Scientific Notation	Meaning
3.78	3.78e0	3.78×10^{0}
37.8	3.78e1	3.78×10^{1}
3780.0	3.78e3	3.78×10^{3}
0.378	3.78e–1	3.78×10^{-1}
0.00378	3.78e–3	3.78×10^{-3}

```
x1 = 10000000000     # 10,000,000,000
type(x1)             # int
x2 = 10e9            # 10,000,000,000.0
type(x2)             # float

eps1 = 0.0001 # 0.0001
eps2 = 1e-4   # 0.0001
```

10 billion written out in full (takes time)

By default scientific notation is always a float

Typical size of an acceptable error of an optimisation process.

Import a Module

Code may be separated into several `.py` files as number of lines of code increases.

Module – is a `.py` file containing Python code. Treat it just as you would a library:

- Access a module through `import` statement
- Use dot notation to access content

`covid.py` and `L1_Code.py`

For example, you have a file `covid.py`:

covid.py

```
"""
Author   : Dr Ekaterina Abramova
Document: Example 'module' written by another developer
"""
#%%
covid19_uk_total = 9.2e+06 # their calculation show this cov19
covid19_uk_deaths = 141e+3 # another variable in the script
```

You can load this `covid.py` file into another script, using `import` command:

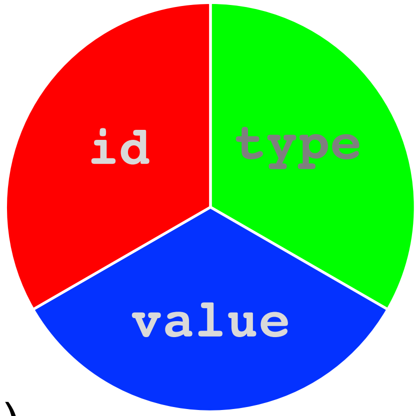
L1_Code.py

```
#%% LOAD MODULE EXAMPLE
# Say in our current script we have a variable:
covid19_uk_total = 9.1e+06
print(covid19_uk_total)

# Load speedOfLight's contents:
import covid
print(covid.covid19_uk_total)

error = covid19_uk_total - covid.covid19_uk_total
print(error)
```

L1_LAB_Questions.py Qn 1

Objects – Formal Definition

Object – contains **3 key elements** which define its overall behavior:

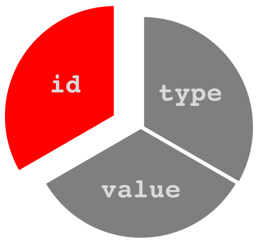


- `identity` – address in memory given by a unique integer
  retrieve using `id()` command – gives back integer representation
  Identity never changes
- `type` – defines operations which are allowed on that object
  check using `type()` command
  Type never changes
- `value` – actual content of the object (representation of quantum of information)
  check value simply using `print()` command
  Value may change depending if object is: mutable or immutable

Example: number `7` representation in Python has 3 key elements:

- **`identity`** : 4423820480 (obtained from using `id(7)` command). This number will be different for each session / machine
- **`type`** : `<class ‘int’>` (obtained with `type(7)` command). Class specifies allowed operations e.g. `+`, `*`, `/`
- **`value`** : 7 obtained with `print(7)` command

Objects - `identity`



Identity – **address in memory** given by an integer. **Identity never changes** once an object is created and remains constant during its lifetime (until you delete the object or start a new session).

- `id()` *function* which outputs the **integer** representing object's identity.
- `is` *operator* which compares identity of 2 variables, results in a **Boolean** `True` or `False`.

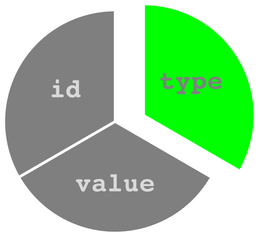
Ways to check if 2 objects point to the same location in memory:

- `id(obj1) == id(obj2)`
- `var1 is var2`

```
# --- IMMUTABLE OBJECT TYPES (e.g. integer)
id(15) # 4423820800 (you'll have a diff num)
id(15) == id(15) # True

x = 15
y = x  # Does not create another object!
id(x)  # 4423820800
id(y)  # 4423820800
x is y # True (i.e. x and y refer to same obj)

z = "abc"
z is y # False
```

```
# --- MUTABLE TYPES (e.g. list)
# Guaranteed to refer to 2 unique, diff lists
L1 = []    # empty list 1
L2 = []    # empty list 2
L1 is L2   # False

# Caution: assigns the same object to both L1 and L2!
L1 = L2 = [] # Assignment on the same line
L1 is L2   # True
```

Objects – `type`



Type – determines **operations** that the object supports and defines **possible values** for objects of that type. **Type does not change** (unless under certain controlled conditions).

`type()` – function provides an object's type (for both built-in & newly created classes/objects).

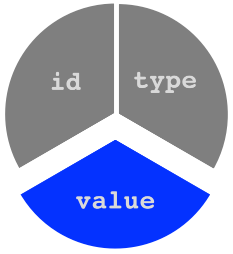
```
In [1]: print(type(1))
<class 'int'>

In [2]: print(type("abc"))
<class 'str'>

In [3]: print(type(5.3))
<class 'float'>

In [4]: print(type(True))
<class 'bool'>
```

Two broad types of objects:

Scalar – indivisible objects that cannot be created from other types (atoms of language).

Non-Scalar – have internal structure.

Objects - `value`



Value – representation of a quantum of information, i.e. object’s state, examples:

- if `x = 5`, then `print(x)` will display the value stored in x.
- relational equality operator == checks for value equality between two objects, if `y = 5`, then `x == y` gives `True`

Value can be changed for some objects, which are divided into two categories:

- **Mutable** objects – those value can be changed **without changing its identity**:

```
myList = [1,2,3]
id(myList)        # 4563681544
myList[0] = 10    # Mutate 1st element
id(myList)        # 4563681544
```

- **Immutable** objects – those value is unchangeable once it is created. **A new object has to be created** if a different value has to be stored.

```
s = "abcdef"
s[0] = "z"
```

Not allowed to change values inside of strings. They are immutable type.

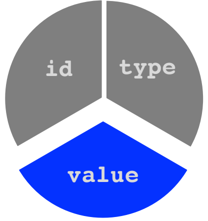
```
In [24]: s[0] = "z"
Traceback (most recent call last):

  File "<ipython-input-24-dabda02f9da1>", line 1, in <module>
    s[0] = "z"

TypeError: 'str' object does not support item assignment
```

Overwriting creates a new object:

```
In [1]: s = "abcdef"

In [2]: id(s)
Out[2]: 140297286192240

In [3]: s = "zbcdef"

In [4]: id(s)
Out[4]: 140297286236208
```

We did not mutate `s`, we created a new object, as evidenced by different `id` number.

Mutating Lists



Methods (i.e. functions) that mutate a list are said to have a '**side effect**'.

Mutate existing list:

1. *Overwriting values*

```
In [1]: L=[1,2,3]

In [2]: id(L)
Out[2]: 4812335624

In [3]: L[0]=10

In [4]: L
Out[4]: [10, 2, 3]

In [5]: id(L)
Out[5]: 4812335624
```

2. *Using methods*

```
In [1]: L=[1,2,3]

In [2]: id(L)
Out[2]: 4827804808

In [3]: L.append(4)

In [4]: L
Out[4]: [1, 2, 3, 4]

In [5]: id(L)
Out[5]: 4827804808
```

Note: id did not change in both cases, therefore this is the same object which has undergone MUTATION.
Method `append()` is said to have a 'side effect'.

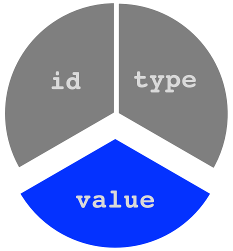
No mutation – new list:

1. *Using operators*

```
In [1]: L1=[1,2,3]

In [2]: id(L1)
Out[2]: 4815391624

In [3]: L2=[4,5,6]

In [4]: id(L2)
Out[4]: 4815891528

In [5]: L3 = L1 + L2

In [6]: L3
Out[6]: [1, 2, 3, 4, 5, 6]

In [7]: id(L3)
Out[7]: 4815858120
```

Note: id has changed, therefore `L3` is a new object – NO MUTATION.

Alias and Mutable Types



Alias – happens when several identifiers reference the same object. E.g.:

Alias for immutable type

```
a = 5
b = a # an alias is created
id(a) == id(b) # True (both identifiers reference the same obj)
```

Alias for mutable type

```
L1 = [1,2,3]
L2 = L1 # alias created
L2 is L1 # True
```

Alias is broken

```
a = 5
b = a
b = 3 # the alias is broken
id(a) == id(b) # False
```

Ways to avoid creating an alias

```
L1 = [1,2,3]
L2 = list(L1) # typecast
L2 is L1      # False
```

```
L1 = [1,2,3]
L2 = [1,2,3] # assign explicitly
L2 is L1 # False
```

Aliasing matters for **mutable types** since object can be **mutated via either path!** Effect of mutation is seen through **BOTH paths.** Great when intended, but difficult to debug. Be careful and aware of this.

Example 1:

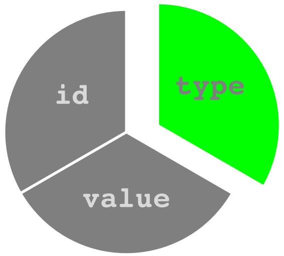
```
L1    = [1,2,3]
L2    = L1       # variables point to the same object
L1[0] = 10       # mutate element 1 in L1
print(L1)        # [10, 2, 3]
print(L2)        # [10, 2, 3]
```

Example 2:

```
L1 = ["LBS", "Imperial", "Oxford"]
L2 = [1,2,3, L1]     # [1, 2, 3, ['LBS', 'Imperial', 'Oxford']]
L1.append("Cambridge")
print(L1)                 # ['LBS', 'Imperial', 'Oxford', 'Cambridge']
print(L2)     # [1, 2, 3, ['LBS', 'Imperial', 'Oxford', 'Cambridge']]
```

L1_LAB_Questions.py Qn 2

Fundamental Data Types



Literals – **built-in objects** in Python. Each literal has a `type` called **fundamental data type**

Data type		<class ‘xxx’>	Mutability	Meaning / Use	Syntax
Scalar	None	`NoneType`	X	Identifies absence of a value	`x=None`

	Numeric	Integers `int`
		`bool`
		Real `float`
			X
			X
			X
				Integers ℤ
				`True` 1, `False` 0 (hence belongs to `int` type)
				Decimals. Double precision floating-point number
					`x=1` `x=(1)`
					`b=True` `b=False` `b=(True)`
					`x=1.` `x=1.0` `x=(1.0)`

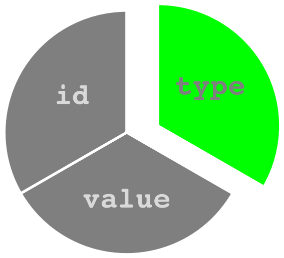
Non-scalar	Sequences	String `str`
		Container `tuple`
		`list`
			X
			X
			✓
				Contiguous set of characters
				Ordered* seq. Elements can be of any type
				Ordered seq. Elements can be of any type
					`s="abc"` `s=("abc")`
					`T=()` `T=(1,)` `T=(1,2,3)`
					`L=[]` `L=[1]` `L=[1,2,3]`

	Sets	Container `frozenset` `set`	X ✓	Unordered unique collection. Elems can be only of immutable type Unordered unique collection. Elems can be only of immutable type	`S=frozenset()` `S=set()` `S={1,2,3}`
	Mappings	Container `dict`	✓	Unordered elems created with `key:value` pair Keys can be any immutable data type; values any type	`D={}` `D={"A":1, "B":2}`

Data structure / Container – object (structure) used to store a collection of data of the same or different types.

! **`str` is considered as both scalar & non-scalar object.**
! Python `int` and `float` are strongly related to mathematical numbers, but are limited to numerical representations in computers.
! Python does not have a `character` type, but supports strings with varied lengths including length one.
* Ordered refers to the order of elements inserted by the user, not sorting of any way.

Type Casting



Type Casting – type conversion between types of built-in objects.

Syntax	From Type	To Type	Examples	Result
`int(x)`	Scalar	integer	`int(5.9), int("5"), int(True)`	5 (i.e. truncates), 5, 1
`float(x)`	Scalar	float	`float(5), float("5"), float("5.5"), float(True)`	`5.0, 5.0, 5.5, 1.0`
`str(x)`	Any	string	`str(None), str(5), str(5.5), str(True), str([1, 2, 3]), str(range(5)), str({"A":1, "B":2, "C":3})`	`'None', '5', '5.5', 'True' '[1,2,3]', 'range(0,5)' "{'A': 1, 'B': 2, 'C': 3}"`
`tuple(x)`	Non-scalar (not `dict`)	tuple	`tuple("Python")` `tuple([1, 2, 3]), tuple(range(5))`	`('P','y','t','h','o','n')` `(1,2,3), (0,1,2,3,4)`
`list(x)`	Non-scalar	list	`list("Python")` `list({1, 2, 3}), list({"A":1, "B":2})`	`['P','y','t','h','o','n']` `[1,2,3], ['A','B']`
`set(x)`	Non-scalar	set	`set("Python")` `set([1, "a"]), set({"A":1, "B":2})`	`{'P','h','n','o','t','y'}` `{1,'a'}, {'A','B'}`
`dict(d)`	Tuple of tuples each with (key, value)	dictionary	`dict( (("a",1), ("b",2)) )` `dict( ((1, "a"), (2, "b")) )`	`{'a':1, 'b':2}` `{1:'a', 2:'b’}`

Open and Read File

Each Operating System (OS) comes with its own file management system. Python achieves OS independence by using **file handles**.

File handle – a temporary internal reference (number) assigned by OS to a file that was requested by a user to be opened. Throughout the session the handle is used to access the file.

Syntax	Example	Use
`open(fileName)`	`f = open("USD.csv")`	Opens an existing file for reading and returns a file handle.

Access methods via **dot notation** `obj.method()`

obj name method name

Syntax	Example	Use
`f.read()`	`f = open("USD.csv")` `s = f.read()`	returns one long **string** with all contents of the file
`f.readline()`	`f = open("USD.csv")` `line = f.readline()`	returns **next line** from the file
`f.readlines()`	`f = open("USD.csv")` `L = f.readlines()`	returns a **list** where each element is 1 line from the file
`f.close()`	`f.close()`	closes file: **always close file** after using it! (Free up memory)

String Methods

Here we will look at some common methods belonging to `<class 'str'>`.

Remember strings are **immutable** type, thus original string remains **unchanged** after an operation. Thus save results to a new variable, i.e. all of the methods do not mutate the string.

Syntax	s ="daddad" s2="abc,DEF"	s is unchanged! ans variable is:	Use
`strip(c)`	`ans = s.strip("d")`	`"adda"`	Removes char c from start and end of `s`. Default `whitespace` (**space, tab, newline**).
`rstrip(c)`	`ans = s.rstrip("d")`	`"dadda"`	**right** removes character c from `s`.
`lstrip(c)`	`ans = s.lstrip("d")`	`"addad"`	**left** removes character c from `s`.
`replace(c,e)`	`ans = s2.replace(",","")`	`"abcDEF"`	Replace character c with character e.
`split(d)`	`ans = s2.split(",")`	`["abc","DEF"]`	Split s using d as delimiter – result is a **`LIST`** Default `whitespace`(**space, tab, newline**).
`lower()`	`ans = s2.lower()`	`"abc,def"`	Converts all upper cases to lower.
`upper()`	`ans = s1.upper()`	`"DADDAD"`	Converts all lower cases to upper.

Further methods are available for string type, which you can see by using **tab completion** **`obj.<TAB>`** i.e. type out object name, then dot, press **`<TAB>`** and all available methods get listed e.g. `s.<TAB>`

String Methods - Example (1)

Read entire file into a single string, using `f.read()`

- Let's use Hillary Clinton's emails dataset (typical in Data Science benchmark work). We will use email subjects (titles) as an example:

```
f = open("Subjects.csv")  # create file handle that references the csv file
s = f.read()              # read in the dataset into a single string s (note this is type string)
f.close()
```

In [1]: s

string: **Out[1]:** 'WOW\nHOW SYRIA IS AIDING QADDAFI\nCHRIS STEVENS\nCAIRO CONDEMNATION - FINAL\n…'

$\underbrace{\text{WOW\n}}_{1}\underbrace{\text{HOW SYRIA IS AIDING QADDAFI\n}}_{2}\underbrace{\text{CHRIS STEVENS\n}}_{3}\underbrace{\text{CAIRO CONDEMNATION - FINAL\n}}_{4}$

Subject number: 1 2 3 4

- Say we want a list that contains one word per element. We can split the string by whitespace (space, tab, \n) using `split()` method, which will do just that:

```
L = s.split()  # note that original string is unchanged (strings are immutable types)
```

In [2]: L

list: **Out[2]:** ['WOW', 'HOW', 'SYRIA', 'IS', 'AIDING', 'QADDAFI', 'CHRIS', 'STEVENS', 'CAIRO', 'CONDEMNATION', 'FINAL',…]

Element / word number: 1 2 3 … 11

String Methods - Example (2)

Read each row of csv as an element of a list, using `f.readlines()`

- The `f.readlines()` will automatically read the csv into a list.

```
f = open("Subjects.csv")  # create file handle that references the csv file
L = f.readlines()         # read in the dataset into a list (each row becomes and element)
f.close()
```

list:

In [1]: L
Out[1]: ['WOW', 'HOW SYRIA IS AIDING QADDAFI', 'CHRIS STEVENS', 'CAIRO CONDEMNATION – FINAL',…]

$\underbrace{\text{'WOW'}}_{1}$, $\underbrace{\text{'HOW SYRIA IS AIDING QADDAFI'}}_{2}$, $\underbrace{\text{'CHRIS STEVENS'}}_{3}$, $\underbrace{\text{'CAIRO CONDEMNATION – FINAL'}}_{4}$

Element / subject number: 1 2 3 4

- We may later choose to work on a particular email subject, e.g. email number 2:

```
subj2 = L[1]           # extract 2nd element of the list (referenced by index 1)
L2 = subj2.split()     # created a list where each word of subject 2 is an element
```

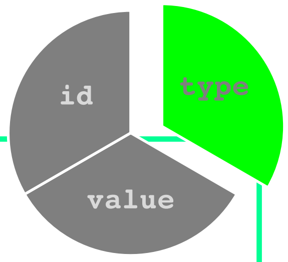
list:

In [2]: L2
Out[2]: ['HOW', 'SYRIA', 'IS', 'AIDING', 'QADDAFI']

Element / word number: 1 2 3 4 5

L1_LAB_Questions.py Qn 3-4

Operations on Sequences



Operations on sequence types: strings, tuples, lists.

Operation	Description
`s[i]`	indexing starts from s[0] not s[1]
`s[i:j]` `s[i:j:step]`	slicing extended slicing
`s1 + s2`	concatenation (sequences of same type)
`s * n`	make `n` copies of `s` (where n is an integer)
`e in s` `e not in s`	membership: element `e` in `s` element `e` not in `s`
`len(s)`	length
`min(s)`	minimum item
`max(s)`	maximum item
`for e in s:`	iteration

Indexing: extract single elem `s[i]`

```
s = ["a", 11, {1,2,3}]
```

position	**1**	**2**	**3**	**How humans think**
index	**0**	**1**	**2**	**Start count left-to-right**
-ve index	**-3**	**-2**	**-1**	**Start count right-to-left**

```
print(s[0])   # 1st elem: "a"
print(s[-1])  # last elem: {1, 2, 3}
s[2] = 15  # mutate last elem ["a", 11, 15]
```

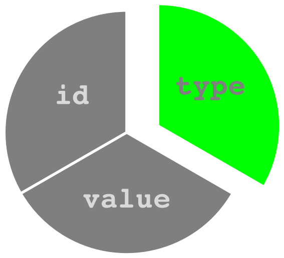
Slicing: extract sub-sequence `s[i:j]`

`s[start:stop]`	`i` to `j-1`
`s[start:]`	`i` to end
`s[:stop]`	0 to `j-1`
`s[:]`	whole seq

Extended slicing: `s[i:j:step]`

`s[start:stop:step]`	`i` to `j-1` with `step`
`s[start::step]`	`i` to end with `step`
`s[::step]`	0 to end with `step`
`s[::]`	whole seq

Slicing Examples




```
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# index:
#     0  1  2  3  4  5  6  7  8  9
# negative index:
#   -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

Note the facts:

- **L[10]** causes an error (cannot index past number of elems)
- **L[9]** allowed and returns the contents of the list at index 9 (**int**)
- **L[9:100]** also works!

Syntax	Description	Indexing
`L[0]` `L[-10]`	**1st element**	Position 1, given by index 0. Position 1, given by index -10

`L[:]`
`L[0:10]`
`L[-10:]`
	All elements
	Means: `L[0:10:1]`
	Means: `L[-10::1]`
		Start & stop omitted, hence whole seq
		0 to 10-1 i.e. index start 0, index stop 9, in steps of 1
		First elem (start indexed by -10) to end, in steps of 1

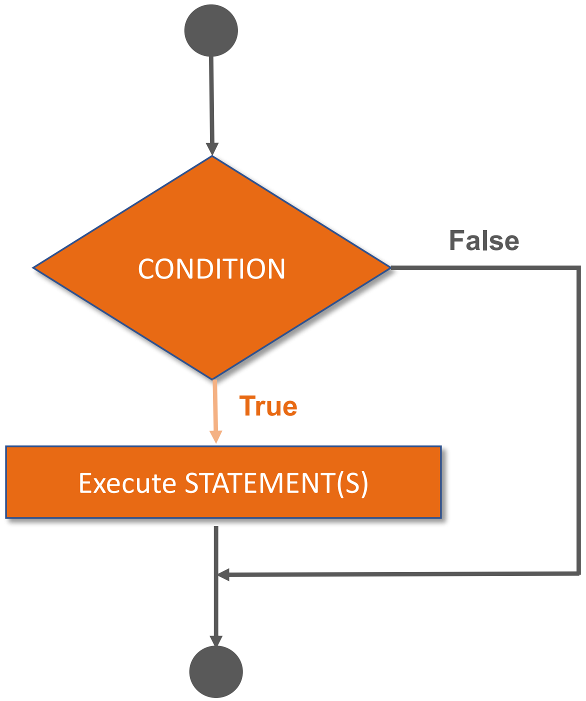
`L[-1]` `L[len(L)-1])`	**Last element**	Convenient definition in Python Using `len()` function would be a lot longer!
`L[::2]`	Every second element	Start to end in steps of 2
`L[::-1]`	**Reverse sequence** Equiv: `L[9::-1]` Equiv: `L[-1::-1]`	Start from last elem since using steps of -1 i.e. from 9 to the beginning in steps of -1 i.e. from -1 to the beginning in steps of -1

`if` - Statement

Straight Line Programming – execute one statement after another in the order they appear.

Branching Programming – branched code execution achieved using CONDITIONAL statements.

`if` statement – execute block of code if `CONDITION` is `True`:



Note use of colon!

```
if CONDITION:
    # block of code
```

Note 4 space (automatic) indentation!

```
x = 5
if x == 5:
    print(x)
```

Block of code executed if `CONDITION` is `True`.

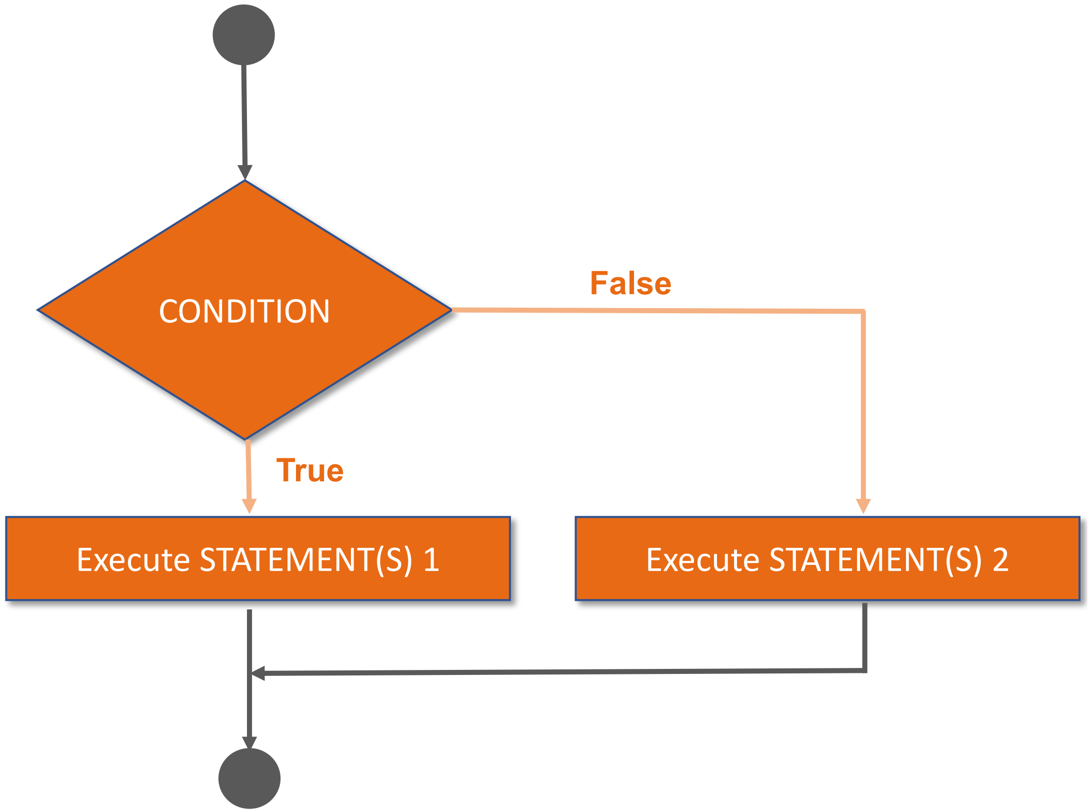
Condition consists of **1 or more** expressions evaluated with relational and/or logical operators – resulting in **`bool`**).

```
x = 5
y = "abc"
if (x == 5) and (y == "abc"):
    print(x, y)
# 5 abc
```

Relational operators:	Logical operators:
`==` `!=` `<` `<=` `>` `>=`	`not` `and` `or`

`if-else` - Statement

- `if-else` – if `CONDITION True` execute block 1, else execute block 2




```
1. if CONDITION:
2.     # block of code 1
3. else:
4.     # block of code 2
```

Note colon!

Note 4 space indentation!

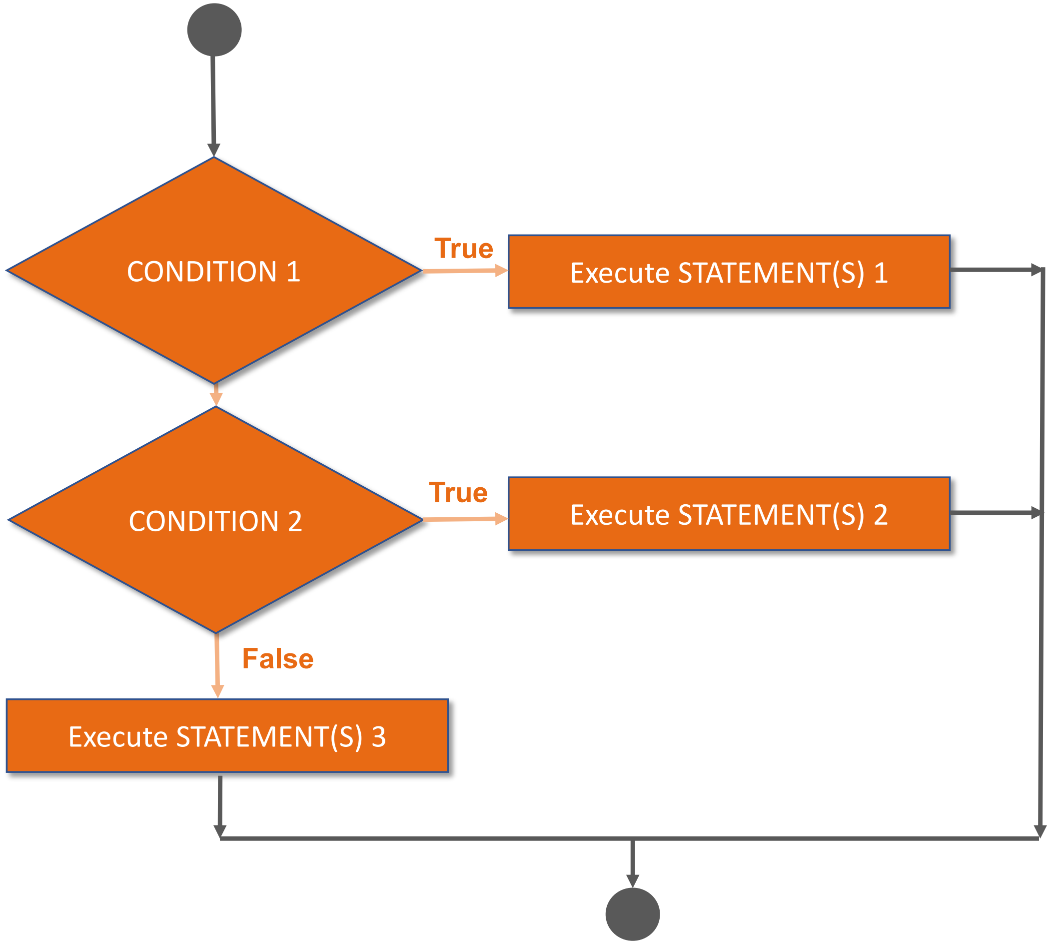
Line up!

Either block 1 or block 2 is **executed**.

```python
x = 7
if x == 5:
    print(x)
else:
    print("x does not equal 5!")
```

if-elif-else - Statement

- if elif else – if CONDITION1 is True execute block 1, else if CONDITION2 is True execute block 2, else execute block 3




```
1.  if CONDITION1:
2.      # block of code 1
3.  elif CONDITION2:
4.      # block of code 2
5.  else:
6.      # block of code 3
```

Note colon!

Line up!

```
x = 10
if x == 5:
    print("x is 5")
elif x == 10:
    print("x is 10")
else:
    print("x is not 5 or 10")
# x is 10
```

Nested `if` Statements

- Nested `if` statements:

```
if CONDITION1:
    # block of code 1
    if CONDITION2:
        # block of code 2
elif CONDITION3:
    # block of code 3
else:
    # block of code 4
```

Note 4 space indentation

Line up!

Treat as a 'block of code'. This block is to be executed only if Condition 1 is `True`.

```
# nested
x = 5; L = ["1","2","3"]
if x == 5:
    print("x is 5")
    if (len(L) != 0):
        for ii in L:
            print(ii)
elif x == 10:
    print("x is 10")
else:
    print("x is not 5 or 10")
# x is 5
# "1"
# "2"
# "3"
```


L1_LAB_Questions.py Qn 5-6

Session Summary

Topics Covered Today:

- Library Management Systems
- Loading Libraries
- Objects (id, type, value)
- Mutation and Aliasing
- Fundamental Data Types
- Operations on Sequences
- Branching Code (`if` statements)

After Each Session:

- Optional Homework
- Optional Datacamp Videos

Please Provide Feedback online using Mentimeter

https://www.menti.com/25rkbpod94

EXTRA SLIDES

`print` Function (1)

- String:

```
print('Hello World')
print("Hello World") # Note: enclosing quoatation marks are not printed
```

- Concatenated strings:

```
x = 13.772
print("String 1 " + "String 2") # Note space is needed after 1
print("Universe is", x, "Billion Years Old!") # Use comma to separate output
print("Age of the universe is: " + str(x)) # Type cast float to string first
```

- Special characters:

```
print('Bank\'s Address') # use \' for single quote inside a 'string'
print("Bank's Address")  # use  ' for single quote inside a "string"
print("Was it a \"great\" idea?") # use \" for double quotes inside a "string"
```

- By default `print()` command **automatically ends with newline character `\n`** i.e. each new `print()` command outputs to new line. To continue printing on the **same line** use (works for consecutive print() commands): **`print(objName, end = " ")`**

```
x = 2
print("Input x is:", x, end=" ")
y = x**2
print("Output y is:", y)
# Input x is: 2 Output y is: 4
```

`print` Function (2)

- Splitting string across lines:

```python
# Splitting across lines is not allowed:
print("This is a very long string that I wish to
        extend all the way to the next line")
# Allowed if use a Doc String:
print("""This is a very long string that I wish to
        extend all the way to the next line""")
# Use newline character \n to place characters following it to next line
print("Contents of first line; \nContents of second line.")
```

- Format objects using `%` inside the `print()` function:

Letter	Type	e.g.
f	float	"%.2f"

```python
x = 15.56755421
print("%.2f" % x) # 15.57
print("%.4f" % x) # 15.5676
```

Format Float – round to `n` decimal pts.
% is the **format operator**, takes form:
`<format string> % <datum>`

Character Sets – `ASCII` / `UNICODE`

- ASCII – American Standard Code for Information Exchange (1980).
- UNICODE – extended ASCII across languages (1990).

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DCI	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	“	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	l	j	k	l	m
11	n	o	P	q	r	S	t	u	v	w
12	X	y	z	{	|	}	~	DEL		

- Characters map to `int` values
- Check position (order) of characters with `ord()` function

```
ord("A")   # 65
ord("B")   # 66
ord("a")   # 97
"A" < "B"  # True
"a" > "A"  # True
```

- Check character at position `x` with `chr()` function:

```
chr(65)    # A
chr(66)    # B
```

`float` - Scientific Notation

- Scientific notation – is used to represent very large/small numbers.

Decimal Notation	Scientific Notation	Meaning
3.78	3.78e0	3.78×10^{0}
37.8	3.78e1	3.78×10^{1}
3780.0	3.78e3	3.78×10^{3}
0.378	3.78e–1	3.78×10^{-1}
0.00378	3.78e–3	3.78×10^{-3}

```
x1 = 10000000000.0 # 10000000000.0
x2 = 1e10          # 10000000000.0

eps1 = 0.0001 # 0.0001
eps2 = 1e-4   # 0.0001
```

Two ways of writing the same number: 10 million (by default always a float).

Typical size of an acceptable error of an optimisation process.

Classes – Terminology Overview

Procedure-oriented programming – evolves around **functions** (re-usable block of code).

Object-oriented programming (OOP) – focuses on **objects**, combines data and functions.

Data abstraction – process of structuring programs to **hide details** of implementation from end-user, thereby making it easier to work with that data.

Encapsulation – (fundamental part of OOP) process of binding data and code (functions) that operate on that data and preventing unauthorised access to them.

Class – abstract collection of data (*attributes*) and functions (*methods*), used as a **blueprint** to create **objects**.

Object – instance of a class (a concrete 'thing' that you made using a specific class). Upon creating an object we specify what values its key data should be equal to.

Classes and objects examples:

- `<class 'human'>` objects: Nick, Jen etc (two different instances of the class human).
- `<class 'house'>` objects: your house, my house. They have different properties: number of rooms etc.
- `<class 'int'>` objects: `2`, `17`. Different properties: number of bits used to represent them in memory).