Data Exploration with Python and Jupyter

Basic usage of the Pandas library to download a dataset, explore its contents, clean up missing or invalid data, filter the data according to different criteria, and plot visualizations of the data.

Press `Spacebar` to go to the next slide (or `?` to see all navigation shortcuts)

Python

is a widely used programming language with many useful libraries

Python

is a widely used programming language with many useful libraries

Jupyter

an interactive notebook style of using a programming language (aka the "Kernel")

Jupyter notebook

Cells

Notebook is separated into cells, which can be

- **code** cells
    - contain Python code to be executed
- **markdown** cells
    - contains text in markdown format

To select a cell: click on it with the mouse

To run the selected cell, click the `Run` button, or press `Ctrl+Enter`, or click "Cell -> Run Cells" on the menubar

In [1]:
```
# This is a code cell: press Ctrl+Enter to execute the code in it
#
print("Hello World!")
```

```
Hello World!
```

In [1]:
```
# This is a code cell: press Ctrl+Enter to execute the code in it
#
print("Hello World!")
```

```
Hello World!
```

Markdown cell

This is a markdown cell, which can contain

- headings, lists, *formatted* **text**, links to websites.
- math in latex format: $\int_0^\infty \cos(x) dx$
- images:

Mode

Two *modes* of interacting with the active/selected cell

- **edit** mode
    - edit the text inside the cell (green outline)
- **command** mode
    - use keyboard shortcuts to modify the cell or run commands (blue outline)
- To enter edit mode: double click inside a cell, or press `Enter` with a cell selected
- To enter command mode: click to the left of a cell inside the green outline, or press `Escape`

Commands

Lots of keyboard shortcuts available. Press `Escape` to enter command mode, then the `H` key to see a list.

Some commonly used shortcuts:

- `A`: insert a cell above the current cell
- `B`: insert a cell below the current cell
- `M`: convert the current cell to a markdown cell
- `Y`: convert the current cell to a code cell
- `Shift+Enter`: run the current cell and advance to the next cell

Order of Execution

- you are free to execute / run cells in any order you choose
- they can make use of and modify any objects, functions or variables that have already been created
- however this can quickly get confusing and make reproducing results difficult!

Top to bottom

- it is good practice to have a top-to-bottom flow of execution
- i.e. write your notebook so that it can be executed in the order it is written
- this makes it easier to understand what is going on

Useful commands when things go wrong

- menubar `Kernel -> Restart` (or command mode shortcut: `0 0`)
    - fresh start (your code is still there, but all existing objects, functions and variables are cleared)
- menubar `Kernel -> Restart and Clear Output`
    - as above, but additionally clears all cell outputs
- menubar `Kernel -> Restart and Run All`
    - as above, but additionally executes all the cells in order

Python: Variables

In [2]:

```
# any lines starting with "#" are comments that Python ignores
#
# assign the number 12 to the variable "a":
#
a = 12
```

In [2]:
```
# any lines starting with "#" are comments that Python ignores
#
# assign the number 12 to the variable "a":
#
a = 12
```

In [3]:
```
# any variable or object can be printed
print(a)
```

```
12
```

In [2]:
```
# any lines starting with "#" are comments that Python ignores
#
# assign the number 12 to the variable "a":
#
a = 12
```

In [3]:
```
# any variable or object can be printed
print(a)
```

```
12
```

In [4]:
```
# display the type of an object
type(a)
```

Out[4]:
```
int
```

In [5]:
```
# variables can be re-assigned, including to different types
a = "Hello!"
```

In [5]:
```
# variables can be re-assigned, including to different types
a = "Hello!"
```

In [6]:
```
print(a)
```

```
Hello!
```

In [5]:
```
# variables can be re-assigned, including to different types
a = "Hello!"
```

In [6]:
```
print(a)
```

```
Hello!
```

In [7]:
```
type(a)
```

Out[7]:
```
str
```

Python: Lists

In [8]:

```
# a list is an ordered container of objects (the objects don't have to h
# create one by listing items inside square brackets, separated by comma
my_list = [1, 3, 88, -13, "hello"]
```

In [8]:
```
# a list is an ordered container of objects (the objects don't have to h
# create one by listing items inside square brackets, separated by comma
my_list = [1, 3, 88, -13, "hello"]
```

In [9]:
```
print(my_list)
```

```
[1, 3, 88, -13, 'hello']
```

In [8]:
```
# a list is an ordered container of objects (the objects don't have to h
# create one by listing items inside square brackets, separated by comma
my_list = [1, 3, 88, -13, "hello"]
```

In [9]:
```
print(my_list)
```

```
[1, 3, 88, -13, 'hello']
```

In [10]:
```
type(my_list)
```

Out[10]:
```
list
```

In [8]:
```
# a list is an ordered container of objects (the objects don't have to h
# create one by listing items inside square brackets, separated by comma
my_list = [1, 3, 88, -13, "hello"]
```

In [9]:
```
print(my_list)
```

```
[1, 3, 88, -13, 'hello']
```

In [10]:
```
type(my_list)
```

Out[10]:
```
list
```

In [11]:
```
len(my_list)
```

Out[11]:
```
5
```

In [12]:
```
# can reference an item in the list by it's index: 0 is the first item
print(my_list[0])
```

```
1
```

In [12]:
```
# can reference an item in the list by it's index: 0 is the first item
print(my_list[0])
```

```
1
```

In [13]:
```
# can also use negative indices: -1 is the last item, -2 the second-to-l
print(my_list[-1])
```

```
hello
```

In [12]:
```
# can reference an item in the list by it's index: 0 is the first item
print(my_list[0])
```

```
1
```

In [13]:
```
# can also use negative indices: -1 is the last item, -2 the second-to-l
print(my_list[-1])
```

```
hello
```

In [14]:
```
# can use slicing to get a subset of the list: here elements with index
print(my_list[1:3])
```

```
[3, 88]
```

In [15]:
```
# can omit starting element of slice: defaults to first element
print(my_list[:2])
```

```
[1, 3]
```

In [15]:
```
# can omit starting element of slice: defaults to first element
print(my_list[:2])
```

```
[1, 3]
```

In [16]:
```
# can omit end element of slice: defaults to number of elements
print(my_list[3:])
```

```
[-13, 'hello']
```

In [15]:
```
# can omit starting element of slice: defaults to first element
print(my_list[:2])
```

```
[1, 3]
```

In [16]:
```
# can omit end element of slice: defaults to number of elements
print(my_list[3:])
```

```
[-13, 'hello']
```

In [17]:
```
# can omit start and end elements of slice: get all elements
print(my_list[:])
```

```
[1, 3, 88, -13, 'hello']
```

In [18]:
```
# can add two lists together: this concatenates them into a single long
print(my_list + [5, 6, 7])
```

```
[1, 3, 88, -13, 'hello', 5, 6, 7]
```

In [18]:
```
# can add two lists together: this concatenates them into a single long
print(my_list + [5, 6, 7])
```

```
[1, 3, 88, -13, 'hello', 5, 6, 7]
```

In [19]:
```
# can iterate over the items in a list
for item in my_list:
    print(item)
```

```
1
3
88
-13
hello
```

Python: Dictionaries

In [20]:

```
# a dictionary is an unordered set of key-value pairs
# create one by listing key:value pairs inside curly brackets, separated
my_dict = {"name": "Bob", "age": 6}
```

In [20]:
```
# a dictionary is an unordered set of key-value pairs
# create one by listing key:value pairs inside curly brackets, separated
my_dict = {"name": "Bob", "age": 6}
```

In [21]:
```
print(my_dict)
```

```
{'name': 'Bob', 'age': 6}
```

In [20]:
```
# a dictionary is an unordered set of key-value pairs
# create one by listing key:value pairs inside curly brackets, separated
my_dict = {"name": "Bob", "age": 6}
```

In [21]:
```
print(my_dict)
```

```
{'name': 'Bob', 'age': 6}
```

In [22]:
```
type(my_dict)
```

Out[22]:
```
dict
```

In [20]:
```
# a dictionary is an unordered set of key-value pairs
# create one by listing key:value pairs inside curly brackets, separated
my_dict = {"name": "Bob", "age": 6}
```

In [21]:
```
print(my_dict)
```

```
{'name': 'Bob', 'age': 6}
```

In [22]:
```
type(my_dict)
```

Out[22]:
```
dict
```

In [23]:
```
len(my_dict)
```

Out[23]:
```
2
```

In [24]:
```
# can look up a value using its key
print(my_dict["name"])
```

```
Bob
```

In [24]:
```
# can look up a value using its key
print(my_dict["name"])
```

```
Bob
```

In [25]:
```
# can add a key-value pair to the dictionary by assinging a value to a k
my_dict["sizes"] = [1, 2, 3]
```

In [24]:
```
# can look up a value using its key
print(my_dict["name"])
```

```
Bob
```

In [25]:
```
# can add a key-value pair to the dictionary by assinging a value to a k
my_dict["sizes"] = [1, 2, 3]
```

In [26]:
```
print(my_dict)
```

```
{'name': 'Bob', 'age': 6, 'sizes': [1, 2, 3]}
```

In [24]:
```
# can look up a value using its key
print(my_dict["name"])
```

```
Bob
```

In [25]:
```
# can add a key-value pair to the dictionary by assinging a value to a k
my_dict["sizes"] = [1, 2, 3]
```

In [26]:
```
print(my_dict)
```

```
{'name': 'Bob', 'age': 6, 'sizes': [1, 2, 3]}
```

In [27]:
```
# adding an existing key overwrites the old value with the new one
my_dict["sizes"] = [5, 10, 24]
```

In [24]:
```
# can look up a value using its key
print(my_dict["name"])
```

```
Bob
```

In [25]:
```
# can add a key-value pair to the dictionary by assinging a value to a k
my_dict["sizes"] = [1, 2, 3]
```

In [26]:
```
print(my_dict)
```

```
{'name': 'Bob', 'age': 6, 'sizes': [1, 2, 3]}
```

In [27]:
```
# adding an existing key overwrites the old value with the new one
my_dict["sizes"] = [5, 10, 24]
```

In [28]:
```
print(my_dict)
```

```
{'name': 'Bob', 'age': 6, 'sizes': [5, 10, 24]}
```

In [29]:
```
# can iterate over dictionary items using dict.items()
for key, value in my_dict.items():
    print(key, value)
```

```
name Bob
age 6
sizes [5, 10, 24]
```

Python: Functions

In [30]:
```
# functions are defined using the def keyword
def my_function():
    print("hi")
```

In [30]:
```
# functions are defined using the def keyword
def my_function():
    print("hi")
```

In [31]:
```
my_function()
```

```
hi
```

In [30]:
```
# functions are defined using the def keyword
def my_function():
    print("hi")
```

In [31]:
```
my_function()
```

```
hi
```

In [32]:
```
# functions can take arguments
def my_function(name):
    print("hi", name)
```

In [30]:
```
# functions are defined using the def keyword
def my_function():
    print("hi")
```

In [31]:
```
my_function()
```

```
hi
```

In [32]:
```
# functions can take arguments
def my_function(name):
    print("hi", name)
```

In [33]:
```
my_function("Liam")
```

```
hi Liam
```

Python: Libraries

In [34]:
```
# import a library, and (optionally) give it a shorter name
import numpy as np
```

In [34]:
```
# import a library, and (optionally) give it a shorter name
import numpy as np
```

In [35]:
```
my_list = [1, 2, 3, 4, 5]
# library functions accessed using library_name.function
# here we create a numpy array from a list
my_array = np.array(my_list)
```

In [34]:
```
# import a library, and (optionally) give it a shorter name
import numpy as np
```

In [35]:
```
my_list = [1, 2, 3, 4, 5]
# library functions accessed using library_name.function
# here we create a numpy array from a list
my_array = np.array(my_list)
```

In [36]:
```
print(my_array)
```

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

In [34]:
```
# import a library, and (optionally) give it a shorter name
import numpy as np
```

In [35]:
```
my_list = [1, 2, 3, 4, 5]
# library functions accessed using library_name.function
# here we create a numpy array from a list
my_array = np.array(my_list)
```

In [36]:
```
print(my_array)
```

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

In [37]:
```
type(my_array)
```

Out[37]:
```
numpy.ndarray
```

In [38]:
```
# apply the numpy `sqrt` function to every element of the array
np.sqrt(my_array)
```

Out[38]:
```
array([1.        , 1.41421356, 1.73205081, 2.        , 2.23606798])
```

In [38]:
```
# apply the numpy `sqrt` function to every element of the array
np.sqrt(my_array)
```

Out[38]:
```
array([1.        , 1.41421356, 1.73205081, 2.        , 2.236067
98])
```

In [39]:
```
# display help about this sqrt function
?np.sqrt
```

In [38]:
```
# apply the numpy `sqrt` function to every element of the array
np.sqrt(my_array)
```

Out[38]:
```
array([1.        , 1.41421356, 1.73205081, 2.        , 2.236067
98])
```

In [39]:
```
# display help about this sqrt function
?np.sqrt
```

In [40]:
```
np.mean(my_array)
```

Out[40]:
```
3.0
```

In [38]:
```
# apply the numpy `sqrt` function to every element of the array
np.sqrt(my_array)
```

Out[38]:
```
array([1.        , 1.41421356, 1.73205081, 2.        , 2.236067
98])
```

In [39]:
```
# display help about this sqrt function
?np.sqrt
```

In [40]:
```
np.mean(my_array)
```

Out[40]:
```
3.0
```

In [41]:
```
np.std(my_array)
```

Out[41]:
```
1.4142135623730951
```

Next

- Part 2: Pandas with toy data