Introduction to Data Science with Python

Arvind Krishna, Lizhen Shi, Emre Besler, and Arend Kuyper

2022-09-20

Table of contents

Preface

This book is developed for the course STAT303-1 (Data Science with Python-1). The first two chapters of the book are a review of python, and will be covered very quickly. Students are expected to know the contents of these chapters beforehand, or be willing to learn it quickly. Students may use the STAT201 book (https://nustat.github.io/Intro_to_programming_for_data_sci/) to review the python basics required for the STAT303 sequence. The core part of the course begins from the third chapter - *Reading data*.

Please feel free to let the instructors know in case of any typos/mistakes/general feedback in this book.

Part I

Quick overview: Python programming

1 Python Basics

This chapter is a very brief introduction to python. If you have not taken STAT201 (Introduction to programming for data science), which is now a pre-requisite for the data science major / minor program, please review the python programming section (chapters 1-6) from the STAT201 book. It is assumed that you are already comfortable with this content. Some of the content of these chapters is reviewed briefly in the first two chapters of this book.

1.1 Python language basics

1.1.1 Object Oriented Programming

Python is an object-oriented programming language. In layman terms, it means that every number, string, data structure, function, class, module, etc., exists in the python interpreter as a python object. An object may have attributes and methods associated with it. For example, let us define a variable that stores an integer:

```
var = 2
```

The variable `var` is an object that has attributes and methods associated with it. For example a couple of its attributes are `real` and `imag`, which store the real and imaginary parts respectively, of the object `var`:

```
print("Real part of 'var': ",var.real)
print("Real part of 'var': ",var.imag)
```

```
Real part of 'var':  2
Real part of 'var':  0
```

Attribute: An attribute is a value associated with an object, defined within the class of the object.

Method: A method is a function associated with an object, defined within the class of the object, and has access to the attributes associated with the object.

For looking at attributes and methods associated with an object, say `obj`, press tab key after typing `obj.`.

Consider the example below of a class *example_class*:

```
class example_class:
    class_name = 'My Class'
    def my_method(self):
        print('Hello World!')

e = example_class()
```

In the above class, `class_name` is an attribute, while `my_method` is a method.

1.1.2 Assigning variable name to object

1.1.2.1 Call by reference

Python utilizes a system, which is known as *Call by Object Reference*. When an object is assigned to a variable name, the variable name serves as a reference to the object. For example, consider the following assignment:

```
x = [5,3]
```

The variable name `x` is a reference to the memory location where the object `[5, 3]` is stored. Now, suppose we assign `x` to a new variable `y`:

```
y = x
```

In the above statement the variable name `y` now refers to the same object `[5,3]`. The object `[5,3]` does **not** get copied to a new memory location referred by `y`. To prove this, let us add an element to `y`:

```
y.append(4)
print(y)
```

```
[5, 3, 4]
```

```
print(x)
```

```
[5, 3, 4]
```

When we changed `y`, note that `x` also changed to the same object, showing that `x` and `y` refer to the same object, instead of referring to different copies of the same object.

1.1.2.2 Assigning multiple variable names

Values can be assigned to multiple variables in a single statement by separating the variable names and values with commas.

```
color1, color2, color3 = "red", "green", "blue"
```

```
color1
```

```
'red'
```

```
color3
```

```
'blue'
```

The same value can be assigned to multiple variables by chaining multiple assignment operations within a single statement.

```
color4 = color5 = color6 = "magenta"
```

1.1.2.3 Rules for variable names

Variable names can be short (`a`, `x`, `y`, etc.) or descriptive (`my_favorite_color`, `profit_margin`, `the_3_musketeers`, etc.). However, we recommend that you use descriptive variable names as it makes it easier to understand the code.

The rules below must be followed while naming Python variables:

- A variable's name must start with a letter or the underscore character `_`. It cannot begin with a number.
- A variable name can only contain lowercase (small) or uppercase (capital) letters, digits, or underscores (`a-z`, `A-Z`, `0-9`, and `_`).
- Variable names are case-sensitive, i.e., `a_variable`, `A_Variable`, and `A_VARIABLE` are all different variables.

Here are some valid variable names:

```
a_variable = 23
is_today_Saturday = False
my_favorite_car = "Delorean"
the_3_musketeers = ["Athos", "Porthos", "Aramis"]
```

Let's try creating some variables with invalid names. Python prints a syntax error if the variable's name is invalid.

> **Syntax**: The syntax of a programming language refers to the rules that govern the structure of a valid instruction or *statement*. If a statement does not follow these rules, Python stops execution and informs you that there is a *syntax error*. Syntax can be thought of as the rules of grammar for a programming language.

```
a variable = 23
```

```
is_today_$aturday = False
```

```
my-favorite-car = "Delorean"
```

```
3_musketeers = ["Athos", "Porthos", "Aramis"]
```

1.1.3 Built-in objects

1.1.3.1 Built-in data types

Variable is created as soon as a value is assigned to it. We don't have to define the type of variable explicitly as in other programming languages because Python can automatically guess the type of data entered **(dynamically typed)**.

Any data or information stored within a Python variable has a *type*. We can view the type of data stored within a variable using the `type` function.

```
a_variable
```

```
23
```

```
type(a_variable)
```

```
int
```

```
is_today_Saturday
```

```
False
```

```
type(is_today_Saturday)
```

```
bool
```

```
my_favorite_car
```

```
'Delorean'
```

```
type(my_favorite_car)
```

```
str
```

```
the_3_musketeers
```

```
['Athos', 'Porthos', 'Aramis']
```

```
type(the_3_musketeers)
```

```
list
```

Python has several built-in data types for storing different kinds of information in variables.

```
<IPython.core.display.Image object>
```

Primitive: Integer, float, boolean, None, and string are *primitive data types* because they represent a single value.

Containers: Other data types like list, tuple, and dictionary are often called *data structures* or *containers* because they hold multiple pieces of data together. We'll discuss these datatypes in chapter 2.

The data type of the object can be identified using the in-built python function `type()`. For example, see the following objects and their types:

```
type(4)
```

```
int
```

```
type(4.4)
```

```
float
```

```
type('4')
```

```
str
```

```
type(True)
```

```
bool
```

1.1.3.2 Built-in modules and functions

Built-in functions in Python are a set of predefined functions that are available for use without the need to import any additional libraries or modules. The Python Standard Library is very extensive. Besides built-in functions, it also contains many Python scripts (with the . py extension) containing useful utilities and modules written in Python that provide standardized solutions for many problems that occur in everyday programming.

Below are a couple of examples:

range(): The `range()` function returns a sequence of evenly-spaced integer values. It is commonly used in `for` loops to define the sequence of elements over which the iterations are performed.

Below is an example where the `range()` function is used to create a sequence of whole numbers upto 10:

```
print(list(range(1,10)))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

Date time: Python as a built-in datetime module for handling date/time objects:

```
import datetime as dt
```

```
#Defining a date-time object
dt_object = dt.datetime(2022, 9, 20, 11,30,0)
```

Information about date and time can be accessed with the relevant attribute of the `datetime` object.

```
dt_object.day
```

```
20
```

```
dt_object.year
```

```
2022
```

The `strftime` method of the `datetime` module formats a `datetime` object as a string. There are several types of formats for representing date as a string:

```
dt_object.strftime('%m/%d/%Y')
```

```
'09/20/2022'
```

```
dt_object.strftime('%m/%d/%y %H:%M')
```

```
'09/20/22 11:30'
```

```
dt_object.strftime('%h-%d-%Y')
```

```
'Sep-20-2022'
```

1.1.4 Importing libraries

There are several built-in functions in python like `print()`, `abs()`, `max()`, `sum()` etc., which do not require importing any library. However, these functions will typically be insufficient for a analyzing data. Some of the popular libraries and their primary purposes are as follows:

1. **NumPy**: NumPy is a fundamental library for numerical computing in Python. It provides support for arrays, matrices, and mathematical functions, making it essential for scientific and data analysis tasks.. It is mostly used for performing numerical operations and efficiently storing numerical data.
2. **Pandas**: Pandas is a powerful data manipulation and analysis library. It offers data structures like DataFrames and Series, which facilitate data reading, cleaning, transformation, and analysis, making it indispensable in data science projects.
3. **Matplotlib, Seaborn**: Matplotlib is a comprehensive library for creating static, animated, or interactive plots and visualizations. It is commonly used for data visualization and exploration in data science. Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.
4. **SciPy**: SciPy is used for performing scientific computing such as solving differential equations, optimization, statistical tests, etc.
5. **Scikit-learn**: Scikit-learn is a machine learning library that provides a wide range of tools for data pre-procesing, classification, regression, clustering, dimensionality reduction, and more. It simplifies the implementation of machine learning algorithms and model evaluation.
6. **Statsmodels**: Statsmodels is used for developing statistical models with a focus on inference *(in contrast to focus on prediction as in scikit-learn)*.

To use libraries like NumPy, pandas, Matplotlib, and scikit-learn in Python, you typically need to follow these steps:

1. Install the libraries (Anaconda already does this)
2. Import the libraries in the python script or jupyter notebook
3. Use the Library Functions and Classes: After importing the libraries, you can use their functions, classes, and methods in your code. For instance, you can create NumPy arrays, manipulate data with pandas, create plots with Matplotlib, or train machine learning models with scikit-learn.

A library can be imported using the `import` keyword. For example, a NumPy library can be imported as:

```
import numpy as np
```

Using the `as` keyboard, the NumPy library has been given the name `np`. All the functions and attributes of the library can be called using the *'np.'* prefix. For example, let us generate a sequence of whole numbers upto `10` using the NumPy function arange():

```
np.arange(8)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

There's different ways to import:

- Import the whole module using its original name: `import math, os`
- Import specific things from the module: `from random import randint from math import pi`
- Import the whole library and rename it, usually using a shorter variable name: `import pandas as pd`
- Import a specific method from the module and rename it as it is imported: `from os.path import join as join_path`

1.1.5 User-defined functions

A function is a reusable set of instructions that takes one or more inputs, performs some operations, and often returns an output. **Indeed, while python's standard library and ecosystem libraries offer a wealth of pre-defined functions for a wide range of tasks, there are situations where defining your own functions is not just beneficial but necessary.**

1.1.5.1 Creating and using functions

You can define a new function using the def keyword.

```
def say_hello():
    print('Hello there!')
    print('How are you?')
```

Note the round brackets or parentheses `()` and colon `:` after the function's name. Both are essential parts of the syntax. The function's *body* contains an indented block of statements.

The statements inside a function's body are not executed when the function is defined. To execute the statements, we need to *call* or *invoke* the function.

```
say_hello()
```

```
Hello there!
How are you?
```

```
def say_hello_to(name):
    print('Hello ', name)
    print('How are you?')
```

```
say_hello_to('Lizhen')
```

```
Hello  Lizhen
How are you?
```

```
name = input ('Please enter your name: ')
say_hello_to(name)
```

```
Please enter your name: George
Hello  George
How are you?
```

1.1.5.2 Variable scope: Local and global Variables

Local variable: When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function. These types of variables are called local variables. For example,

```
def greet():
    message = 'Hello'  # local variable
    print('Local', message)
greet()
```

```
Local Hello
```

```
print(message) # try to access message variable outside greet() function
```

```
NameError: name 'message' is not defined
```

As `message` was defined within the function `greet()`, it is local to the function, and cannot be called outside the function.

Global variable: Aa variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created.

```
message = 'Hello'  # declare global variable

def greet():
    print('Local', message)  # declare local variable

greet()
print('Global', message)
```

```
Local Hello
Global Hello
```

1.1.5.3 Named arguments

Invoking a function with many arguments can often get confusing and is prone to human errors. Python provides the option of invoking functions with *named* arguments for better clarity. You can also split function invocation into multiple lines.

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    return emi
```

```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12,
    down_payment=3e5
```

```
)
```

```
emi1
```

```
14567.19753389219
```

1.1.5.4 Optional Arguments

Functions with optional arguments offer more flexibility in how you can use them. You can call the function with or without the argument, and if there is no argument in the function call, then a default value is used.

```
emi2 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12)

emi2
```

```
19119.4467632335
```

1.1.5.5 *args and **kwargs

We can pass a variable number of arguments to a function using special symbols. There are two special symbols:

Special Symbols Used for passing arguments: 1. *args (Non-Keyword Arguments) 2. *$kwargs$ *(Keyword Arguments) > Note: "We use the "wildcard" or "" notation like this – *args OR **kwargs – as our function's argument when we have doubts about the number of arguments we should pass in a function."

```
def myFun(*args,**kwargs):
    print("args: ", args)
    print("kwargs: ", kwargs)


# Now we can use both *args ,**kwargs
# to pass arguments to this function :
myFun('John',22,'cs',name="John",age=22,major="cs")
```

```
args:  ('John', 22, 'cs')
kwargs:  {'name': 'John', 'age': 22, 'major': 'cs'}
```

1.1.6 Branching and looping (control flow)

As in other languages, python has built-in keywords that provide conditional flow of control in the code.

1.1.6.1 Branching with `if`, `else` and `elif`

One of the most powerful features of programming languages is *branching*: the ability to make decisions and execute a different set of statements based on whether one or more conditions are true.

The `if` statement

In Python, branching is implemented using the `if` statement, which is written as follows:

```
if condition:
    statement1
    statement2
```

The `condition` can be a value, variable or expression. If the condition evaluates to `True`, then the statements within the *`if` block* are executed. Notice the four spaces before `statement1`, `statement2`, etc. The spaces inform Python that these statements are associated with the `if` statement above. This technique of structuring code by adding spaces is called *indentation*.

> **Indentation**: Python relies heavily on *indentation* (white space before a statement) to define code structure. This makes Python code easy to read and understand. You can run into problems if you don't use indentation properly. Indent your code by placing the cursor at the start of the line and pressing the `Tab` key once to add 4 spaces. Pressing `Tab` again will indent the code further by 4 more spaces, and press `Shift+Tab` will reduce the indentation by 4 spaces.

For example, let's write some code to check and print a message if a given number is even.

```
a_number = 34
```

```
if a_number % 2 == 0:
    print("We're inside an if block")
    print('The given number {} is even.'.format(a_number))
```

```
We're inside an if block
The given number 34 is even.
```

The `else` statement

We may want to print a different message if the number is not even in the above example. This can be done by adding the `else` statement. It is written as follows:

```
if condition:
    statement1
    statement2
else:
    statement4
    statement5
```

If `condition` evaluates to `True`, the statements in the `if` block are executed. If it evaluates to `False`, the statements in the `else` block are executed.

```
if a_number % 2 == 0:
    print('The given number {} is even.'.format(a_number))
else:
    print('The given number {} is odd.'.format(a_number))
```

```
The given number 34 is even.
```

The `elif` statement

Python also provides an `elif` statement (short for "else if") to chain a series of conditional blocks. The conditions are evaluated one by one. For the first condition that evaluates to `True`, the block of statements below it is executed. The remaining conditions and statements are not evaluated. So, in an `if`, `elif`, `elif`… chain, at most one block of statements is executed, the one corresponding to the first condition that evaluates to `True`.

```
today = 'Wednesday'
```

```
if today == 'Sunday':
    print("Today is the day of the sun.")
elif today == 'Monday':
    print("Today is the day of the moon.")
elif today == 'Tuesday':
    print("Today is the day of Tyr, the god of war.")
```

```
elif today == 'Wednesday':
    print("Today is the day of Odin, the supreme diety.")
elif today == 'Thursday':
    print("Today is the day of Thor, the god of thunder.")
elif today == 'Friday':
    print("Today is the day of Frigga, the goddess of beauty.")
elif today == 'Saturday':
    print("Today is the day of Saturn, the god of fun and feasting.")
```

```
Today is the day of Odin, the supreme diety.
```

In the above example, the first 3 conditions evaluate to `False`, so none of the first 3 messages are printed. The fourth condition evaluates to `True`, so the corresponding message is printed. The remaining conditions are skipped. Try changing the value of `today` above and re-executing the cells to print all the different messages.

Using `if`, `elif`, and `else` together

You can also include an `else` statement at the end of a chain of `if`, `elif`… statements. This code within the `else` block is evaluated when none of the conditions hold true.

```
a_number = 49
```

```
if a_number % 2 == 0:
    print('{} is divisible by 2'.format(a_number))
elif a_number % 3 == 0:
    print('{} is divisible by 3'.format(a_number))
elif a_number % 5 == 0:
    print('{} is divisible by 5'.format(a_number))
else:
    print('All checks failed!')
    print('{} is not divisible by 2, 3 or 5'.format(a_number))
```

```
All checks failed!
49 is not divisible by 2, 3 or 5
```

Non-Boolean Conditions

Note that conditions do not necessarily have to be booleans. In fact, a condition can be any value. The value is converted into a boolean automatically using the `bool` operator. Any value in Python can be converted to a Boolean using the `bool` function.

Only the following values evaluate to `False` (they are often called *falsy* values):

1. The value `False` itself
2. The integer `0`
3. The float `0.0`
4. The empty value `None`
5. The empty text `""`
6. The empty list `[]`
7. The empty tuple `()`
8. The empty dictionary `{}`
9. The empty set `set()`
10. The empty range `range(0)`

Everything else evaluates to `True` (a value that evaluates to `True` is often called a *truthy* value).

```
if '':
    print('The condition evaluted to True')
else:
    print('The condition evaluted to False')
```

```
The condition evaluted to False
```

```
if 'Hello':
    print('The condition evaluted to True')
else:
    print('The condition evaluted to False')
```

```
The condition evaluted to True
```

```
if { 'a': 34 }:
    print('The condition evaluted to True')
else:
    print('The condition evaluted to False')
```

```
The condition evaluted to True
```

```
if None:
    print('The condition evaluted to True')
else:
    print('The condition evaluted to False')
```

```
The condition evaluted to False
```

Nested conditional statements

The code inside an `if` block can also include an `if` statement inside it. This pattern is called `nesting` and is used to check for another condition after a particular condition holds true.

```
a_number = 15
```

```
if a_number % 2 == 0:
    print("{} is even".format(a_number))
    if a_number % 3 == 0:
        print("{} is also divisible by 3".format(a_number))
    else:
        print("{} is not divisibule by 3".format(a_number))
else:
    print("{} is odd".format(a_number))
    if a_number % 5 == 0:
        print("{} is also divisible by 5".format(a_number))
    else:
        print("{} is not divisibule by 5".format(a_number))
```

```
15 is odd
15 is also divisible by 5
```

Notice how the `print` statements are indented by 8 spaces to indicate that they are part of the inner `if`/`else` blocks.

> Nested `if`, `else` statements are often confusing to read and prone to human error. It's good to avoid nesting whenever possible, or limit the nesting to 1 or 2 levels.

Shorthand `if` conditional expression

A frequent use case of the `if` statement involves testing a condition and setting a variable's value based on the condition.

Python provides a shorter syntax, which allows writing such conditions in a single line of code. It is known as a *conditional expression*, sometimes also referred to as a *ternary operator*. It has the following syntax:

```
x = true_value if condition else false_value
```

It has the same behavior as the following `if-else` block:

```
if condition:
    x = true_value
else:
    x = false_value
```

Let's try it out for the example above.

```
parity = 'even' if a_number % 2 == 0 else 'odd'
```

```
print('The number {} is {}.'.format(a_number, parity))
```

```
The number 15 is odd.
```

The pass statement

`if` statements cannot be empty, there must be at least one statement in every `if` and `elif` block. We can use the `pass` statement to do nothing and avoid getting an error.

```
a_number = 9
```

```
if a_number % 2 == 0:

elif a_number % 3 == 0:
    print('{} is divisible by 3 but not divisible by 2')
```

```
IndentationError: expected an indented block (1562158884.py, line 3)
```

As there must be at least one statement withihng the `if` block, the above code throws an error.

```
if a_number % 2 == 0:
    pass
elif a_number % 3 == 0:
    print('{} is divisible by 3 but not divisible by 2'.format(a_number))
```

```
9 is divisible by 3 but not divisible by 2
```

1.1.6.2 Iteration with `while` loops

Another powerful feature of programming languages, closely related to branching, is running one or more statements multiple times. This feature is often referred to as *iteration* on *looping*, and there are two ways to do this in Python: using `while` loops and `for` loops.

`while` loops have the following syntax:

```
while condition:
    statement(s)
```

Statements in the code block under `while` are executed repeatedly as long as the `condition` evaluates to `True`. Generally, one of the statements under `while` makes some change to a variable that causes the condition to evaluate to `False` after a certain number of iterations.

Let's try to calculate the factorial of `100` using a `while` loop. The factorial of a number `n` is the product (multiplication) of all the numbers from `1` to `n`, i.e., `1*2*3*...*(n-2)*(n-1)*n`.

```
result = 1
i = 1

while i <= 10:
    result = result * i
    i = i+1

print('The factorial of 100 is: {}'.format(result))
```

```
The factorial of 100 is: 3628800
```

1.1.6.3 Infinite Loops

Suppose the condition in a `while` loop always holds true. In that case, Python repeatedly executes the code within the loop forever, and the execution of the code never completes. This situation is called an infinite loop. It generally indicates that you've made a mistake in your code. For example, you may have provided the wrong condition or forgotten to update a variable within the loop, eventually falsifying the condition.

If your code is *stuck* in an infinite loop during execution, just press the "Stop" button on the toolbar (next to "Run") or select "Kernel > Interrupt" from the menu bar. This will *interrupt* the execution of the code. The following two cells both lead to infinite loops and need to be interrupted.

```
# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i <= 100:
    result = result * i
    # forgot to increment i
```

```
# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i > 0 : # wrong condition
    result *= i
    i += 1
```

1.1.6.4 `break` and `continue` statements

In Python, `break` and `continue` statements can alter the flow of a normal loop.

We can use the `break` statement within the loop's body to immediately stop the execution and *break* out of the loop. with the continue statement. If the condition evaluates to `True`, then the loop will move to the next iteration.

```
i = 1
result = 1

while i <= 100:
    result *= i
    if i == 42:
        print('Magic number 42 reached! Stopping execution..')
        break
    i += 1

print('i:', i)
print('result:', result)
```

```
Magic number 42 reached! Stopping execution..
i: 42
```

```
result: 14050061177528798985431426062445115699363840000000000
```

```
i = 1
result = 1

while i < 8:
    i += 1
    if i % 2 == 0:
        print('Skipping {}'.format(i))
        continue
    print('Multiplying with {}'.format(i))
    result = result * i

print('i:', i)
print('result:', result)
```

```
Skipping 2
Multiplying with 3
Skipping 4
Multiplying with 5
Skipping 6
Multiplying with 7
Skipping 8
i: 8
result: 105
```

In the example above, the statement `result = result * i` inside the loop is skipped when `i` is even, as indicated by the messages printed during execution.

> **Logging**: The process of adding `print` statements at different points in the code *(often within loops and conditional statements)* for inspecting the values of variables at various stages of execution is called logging. As our programs get larger, they naturally become prone to human errors. Logging can help in verifying the program is working as expected. In many cases, `print` statements are added while writing & testing some code and are removed later.

Task: Guess the output and explain it.

```
# Use of break statement inside the loop

for val in "string":
    if val == "i":
```

```
            break
        print(val)

    print("The end")
```

```
s
t
r
The end
```

```
    # Program to show the use of continue statement inside loops

    for val in "string":
        if val == "i":
            continue
        print(val)

    print("The end")
```

```
s
t
r
n
g
The end
```

1.1.6.5 Iteration with `for` loops

A `for` loop is used for iterating or looping over sequences, i.e., lists, tuples, dictionaries, strings, and *ranges*. For loops have the following syntax:

```
for value in sequence:
    statement(s)
```

The statements within the loop are executed once for each element in `sequence`. Here's an example that prints all the element of a list.

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for day in days:
    print(day)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
```

```
# Looping over a string
for char in 'Monday':
    print(char)
```

```
M
o
n
d
a
y
```

```
# Looping over a dictionary
person = {
    'name': 'John Doe',
    'sex': 'Male',
    'age': 32,
    'married': True
}

for key, value in person.items():
    print("Key:", key, ",", "Value:", value)
```

```
Key: name , Value: John Doe
Key: sex , Value: Male
Key: age , Value: 32
Key: married , Value: True
```

1.1.7 Iterating using `range` and `enumerate`

The `range` function is used to create a sequence of numbers that can be iterated over using a `for` loop. It can be used in 3 ways:

- `range(n)` - Creates a sequence of numbers from `0` to `n-1`
- `range(a, b)` - Creates a sequence of numbers from `a` to `b-1`
- `range(a, b, step)` - Creates a sequence of numbers from `a` to `b-1` with increments of `step`

Let's try it out.

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

```
for i in range(3, 8):
    print(i)
```

```
3
4
5
6
7
```

```
for i in range(3, 14, 4):
    print(i)
```

```
3
7
11
```

Ranges are used for iterating over lists when you need to track the index of elements while iterating.

```
a_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for i in range(len(a_list)):
    print('The value at position {} is {}.'.format(i, a_list[i]))
```

```
The value at position 0 is Monday.
The value at position 1 is Tuesday.
The value at position 2 is Wednesday.
The value at position 3 is Thursday.
The value at position 4 is Friday.
```

Another way to achieve the same result is by using the `enumerate` function with `a_list` as an input, which returns a tuple containing the index and the corresponding element.

```
for i, val in enumerate(a_list):
    print('The value at position {} is {}.'.format(i, val))
```

```
The value at position 0 is Monday.
The value at position 1 is Tuesday.
The value at position 2 is Wednesday.
The value at position 3 is Thursday.
The value at position 4 is Friday.
```

2 Data structures

In this chapter we'll learn about the python data structures that are often used or appear while analyzing data.

2.1 Tuple

Tuple is a sequence of python objects, with two key characteristics: (1) the number of objects are fixed, and (2) the objects are immutable, i.e., they cannot be changed.

Tuple can be defined as a sequence of python objects separated by commas, and enclosed in rounded brackets (). For example, below is a tuple containing three integers.

```
tuple_example = (2,7,4)
```

We can check the data type of a python object using the in-built python function *type()*. Let us check the data type of the object *tuple_example*.

```
type(tuple_example)
```

```
tuple
```

2.1.1 Tuple Indexing

Tuple is ordered, meaning you can access specific elements in a list using their index. Indexing in lists includes both positive indexing (starting from 0 for the first element) and negative indexing (starting from -1 for the last element).

Elements of a tuple can be extracted using their index within square brackets. For example the second element of the tuple *tuple_example* can be extracted as follows:

```
tuple_example[1]
```

```
7
```

```
tuple_example[-1]
```

```
4
```

Note that an element of a tuple cannot be modified. For example, consider the following attempt in changing the second element of the tuple *tuple_example*.

```
tuple_example[1] = 8
```

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

The above code results in an error as tuple elements cannot be modified.

2.1.2 Concatenating tuples

Tuples can be concatenated using the + operator to produce a longer tuple:

```
(2,7,4) + ("another", "tuple") + ("mixed","datatypes",5)
```

```
(2, 7, 4, 'another', 'tuple', 'mixed', 'datatypes', 5)
```

Multiplying a tuple by an integer results in repetition of the tuple:

```
(2,7,"hi") * 3
```

```
(2, 7, 'hi', 2, 7, 'hi', 2, 7, 'hi')
```

2.1.3 Unpacking tuples

If tuples are assigned to an expression containing multiple variables, the tuple will be unpacked and each variable will be assigned a value as per the order in which it appears. See the example below.

```
x,y,z  = (4.5, "this is a string", (("Nested tuple",5)))
```

```
x
```

```
4.5
```

```
y
```

```
'this is a string'
```

```
z
```

```
('Nested tuple', 5)
```

If we are interested in retrieving only some values of the tuple, the expression *_ can be used to discard the other values. Let's say we are interested in retrieving only the first and the last two values of the tuple:

```
x,*_,y,z  = (4.5, "this is a string", (("Nested tuple",5)),"99",99)
```

```
x
```

```
4.5
```

```
y
```

```
'99'
```

```
z
```

```
99
```

2.1.4 Tuple methods

A couple of useful tuple methods are `count`, which counts the occurrences of an element in the tuple and `index`, which returns the position of the first occurrence of an element in the tuple:

```
tuple_example = (2,5,64,7,2,2)
```

```
tuple_example.count(2)
```

```
3
```

```
tuple_example.index(2)
```

```
0
```

Now that we have an idea about tuple, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

2.2 List

List is a sequence of python objects, with two key characeterisics that differentiates it from tuple: (1) the number of objects are variable, i.e., objects can be added or removed from a list, and (2) the objects are mutable, i.e., they can be changed.

List can be defined as a sequence of python objects separated by commas, and enclosed in square brackets []. For example, below is a list consisting of three integers.

```
list_example = [2,7,4]
```

List indexing works the same way as tuple indexing.

2.2.1 Slicing a list

List slicing is a technique in Python that allows you to extract a portion of a list by specifying a range of indices. It creates a new list containing the elements from the original list within that specified range. List slicing uses the colon : operator to indicate the start, stop, and step values for the slice. The general syntax is: `new_list = original_list[start:stop:step]`

Here's what each part of the slice means: * **start**: The index at which the slice begins (inclusive). If omitted, it starts from the beginning (index 0). * **stop**: The index at which the slice ends (exclusive). If omitted, it goes until the end of the list. * **step**: The interval between elements in the slice. If omitted, it defaults to 1.

```
list_example6 = [4,7,3,5,7,1,5,87,5]
```

Let us extract a slice containing all the elements from the the 3rd position to the 7th position.

```
list_example6[2:7]
```

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

Note that while the element at the `start` index is included, the element with the `stop` index is excluded in the above slice.

If either the `start` or `stop` index is not mentioned, the slicing will be done from the beginning or until the end of the list, respectively.

```
list_example6[:7]
```

```
[4, 7, 3, 5, 7, 1, 5]
```

```
list_example6[2:]
```

```
[3, 5, 7, 1, 5, 87, 5]
```

To slice the list relative to the end, we can use negative indices:

```
list_example6[-4:]
```

```
[1, 5, 87, 5]
```

```
list_example6[-4:-2:]
```

```
[1, 5]
```

An extra colon (':') can be used to slice every nth element of a list.

```
#Selecting every 3rd element of a list
list_example6[::3]
```

```
[4, 5, 5]
```

```
#Selecting every 3rd element of a list from the end
list_example6[::-3]
```

```
[5, 1, 3]
```

```
#Selecting every element of a list from the end or reversing a list
list_example6[::-1]
```

```
[5, 87, 5, 1, 7, 5, 3, 7, 4]
```

2.2.2 Adding and removing elements in a list

We can add elements at the end of the list using the *append* method. For example, we append the string 'red' to the list *list_example* below.

```
list_example.append('red')
```

```
list_example
```

```
[2, 7, 4, 'red']
```

Note that the objects of a list or a tuple can be of different datatypes.

An element can be added at a specific location of the list using the *insert* method. For example, if we wish to insert the number 2.32 as the second element of the list *list_example*, we can do it as follows:

```
list_example.insert(1,2.32)
```

```
list_example
```

```
[2, 2.32, 7, 4, 'red']
```

For removing an element from the list, the *pop* and *remove* methods may be used. The *pop* method removes an element at a particular index, while the *remove* method removes the element's first occurence in the list by its value. See the examples below.

Let us say, we need to remove the third element of the list.

```
list_example.pop(2)
```

```
7
```

```
list_example
```

```
[2, 2.32, 4, 'red']
```

Let us say, we need to remove the element 'red'.

```
list_example.remove('red')
```

```
list_example
```

```
[2, 2.32, 4]
```

```
#If there are multiple occurrences of an element in the list, the first occurence will be
list_example2 = [2,3,2,4,4]
list_example2.remove(2)
list_example2
```

```
[3, 2, 4, 4]
```

For removing multiple elements in a list, either `pop` or `remove` can be used in a `for` loop, or a `for` loop can be used with a condition. See the examples below.

Let's say we need to remove intergers less than 100 from the following list.

```
list_example3 = list(range(95,106))
list_example3
```

```
[95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105]
```

```
#Method 1: For loop with remove, iterating over the elements of the original list,
#but updating a copy of the original list
```

```
list_example3_filtered = list(list_example3) #
for element in list_example3:
    if element<100:
        list_example3_filtered.remove(element)
print(list_example3_filtered)
```

```
[100, 101, 102, 103, 104, 105]
```

Q1: What's the need to define a new variable `list_example3_filtered` in the above code?

A1: Replace list_example3_filtered with list_example3 and identify the issue. After an element is removed from the list, all the elements that come afterward have their index/position reduced by one. After the elment `95` is removed, `96` is at index `0`, but the for loop will now look at the element at index `1`, which is now `97`. So, iterating over the same list that is being updated in the loop will keep 96 and 98. Using a new list gets rid of the issue by keeping the original list unchanged, so the for-loop iterates over all elements of the original list.

Another method could have been to interate over a copy of the original list and update the original list as shown below.

```
#Method 2: For loop with remove, iterating over the elements of a copy of the original lis
#but updating the original list
for element in list_example3[:]: #Slicing a list creates a new list, thus the loop is iter
    if element<100:
        list_example3.remove(element)
print(list_example3)
```

```
[100, 101, 102, 103, 104, 105]
```

Below is another method that uses a shorthand notation - list comprehension (explained in the next section).

```
#Method 3: For loop with condition in list comprehension
list_example3 = list(range(95,106))
[element for element in list_example3 if element>=100]
```

```
[100, 101, 102, 103, 104, 105]
```

2.2.3 List comprehensions

List comprehensions provide a concise and readable way to create new lists by applying an expression to each item in an iterable (e.g., a list, tuple, or range) and optionally filtering the items based on a condition. They are a powerful and efficient way to generate lists without the need for explicit loops. The basic syntax of a list comprehension is as follows:

```
new_list = [expression for item in iterable if condition]
```

- **expression**: This is the expression that is applied to each item in the iterable. It defines what will be included in the new list.
- **item**: This is a variable that represents each element in the iterable as the comprehension iterates through it.
- **iterable**: This is the source from which the elements are taken. It can be any iterable, such as a list, tuple, range, or other iterable objects.
- **condition (optional)**: This is an optional filter that can be applied to control which items from the iterable are included in the new list. If omitted, all items from the iterable are included.

Example: Create a list that has squares of natural numbers from 5 to 15.

```
sqrt_natural_no_5_15 = [(x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

```
[25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

Example: Create a list of tuples, where each tuple consists of a natural number and its square, for natural numbers ranging from 5 to 15.

```
sqrt_natural_no_5_15 = [(x,x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

```
[(5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100), (11, 121), (12, 144), (13, 169), (14
```

Example: Creating a list of words that start with the letter 'a' in a given list of words.

```
words = ['apple', 'banana', 'avocado', 'grape', 'apricot']
a_words = [word for word in words if word.startswith('a')]
print(a_words)
```

```
['apple', 'avocado', 'apricot']
```

Example: Create a list of even numbers from 1 to 20.

```
even_numbers = [x for x in range(1, 21) if x % 2 == 0]
print(even_numbers)
```

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

List comprehensions are not only concise but also considered more Pythonic and often more efficient than using explicit loops for simple operations. They can make your code cleaner and easier to read, especially for operations that transform or filter data in a list.

2.2.4 Practice exercise 1

Below is a list consisting of responses to the question: "At what age do you think you will marry?" from students of the STAT303-1 Fall 2022 class.

```
exp_marriage_age=['24','30','28','29','30','27','26','28','30+','26','28','30','30','30','
```

Use list comprehension to:

2.2.4.1

Remove the elements that are not integers - such as *'probably never', '30+'*, etc. What is the length of the new list?

Hint: The built-in python function of the `str` class - isdigit() may be useful to check if the string contains only digits.

Solution:

```
exp_marriage_age_num = [x for x in exp_marriage_age if x.isdigit()==True]
print("Length of the new list = ",len(exp_marriage_age_num))
```

```
Length of the new list =  181
```

2.2.4.2

Cap the values greater than 80 to 80, in the clean list obtained in (1). What is the mean age when people expect to marry in the new list?

```
exp_marriage_age_capped = [min(int(x),80) for x in exp_marriage_age_num]
print("Mean age when people expect to marry = ", sum(exp_marriage_age_capped)/len(exp_marr
```

```
Mean age when people expect to marry =  28.955801104972377
```

2.2.4.3

Determine the percentage of people who expect to marry at an age of 30 or more.

```
print("Percentage of people who expect to marry at an age of 30 or more =", str(100*sum([1
```

```
Percentage of people who expect to marry at an age of 30 or more = 37.01657458563536 %
```

2.2.4.4

Redo Q2.2.4.2 using the if-else statement within list comprehension.

2.2.5 Practice exercise 2

Below is a list consisting of responses to the question: "What do you expect your starting salary to be after graduation, to the nearest thousand dollars? (ex: 47000)" from students of the STAT303-1 Fall 2023. class.

```
expected_salary = ['90000', '110000', '100000', '90k', '80000', '47000', '100000', '70000'
```

Clean `expected_salary` using list comprehensions only, and find the mean expected salary.

2.2.6 Concatenating lists

As in tuples, lists can be concatenated using the `+` operator:

```
import time as tm
```

```
list_example4 = [5,'hi',4]
list_example4 = list_example4 + [None,'7',9]
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

For adding elements to a list, the `extend` method is preferred over the `+` operator. This is because the `+` operator creates a new list, while the `extend` method adds elements to an existing list. Thus, the `extend` operator is more memory efficient.

```
list_example4 = [5,'hi',4]
list_example4.extend([None, '7', 9])
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

2.2.7 Sorting a list

A list can be sorted using the `sort` method:

```
list_example5 = [6,78,9]
list_example5.sort(reverse=True) #the reverse argument is used to specify if the sorting i
list_example5
```

```
[78, 9, 6]
```

2.2.8 Practice exercise 3

Start with the list [8,9,10]. Do the following:

2.2.8.1

Set the second entry (index 1) to 17

```
L = [8,9,10]
L[1]=17
```

2.2.8.2

Add 4, 5, and 6 to the end of the list

```
L = L+[4,5,6]
```

2.2.8.3

Remove the first entry from the list

```
L.pop(0)
```

```
8
```

2.2.8.4

Sort the list

```
L.sort()
```

2.2.8.5

Double the list (concatenate the list to itself)

```
L=L+L
```

2.2.8.6

Insert 25 at index 3

The final list should equal [4,5,6,25,10,17,4,5,6,10,17]

```
L.insert(3,25)
L
```

```
[4, 5, 6, 25, 10, 17, 4, 5, 6, 10, 17]
```

Now that we have an idea about lists, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

2.2.9 Other list operations

You can test whether a list contains a value using the `in` operator.

```
list_example6
```

```
[4, 7, 3, 5, 7, 1, 5, 87, 5]
```

```
6 in list_example6
```

```
False
```

```
7 in list_example6
```

```
True
```

2.2.10 Lists: methods

Just like strings, there are several in-built methods to manipulate a list. However, unlike strings, most list methods modify the original list rather than returning a new one. Here are some common list operations:

2.2.11 Lists vs tuples

Now that we have learned about lists and tuples, let us compare them.

Q2: A list seems to be much more flexible than tuple, and can replace a tuple almost everywhere. Then why use tuple at all?

A2: The additional flexibility of a list comes at the cost of efficiency. Some of the advantages of a tuple over a list are as follows:

1. Since a list can be extended, space is over-allocated when creating a list. A tuple takes less storage space as compared to a list of the same length.

2. Tuples are not copied. If a tuple is assigned to another tuple, both tuples point to the same memory location. However, if a list is assigned to another list, a new list is created consuming the same memory space as the original list.
3. Tuples refer to their element directly, while in a list, there is an extra layer of pointers that refers to their elements. Thus it is faster to retrieve elements from a tuple.

The examples below illustrate the above advantages of a tuple.

```
#Example showing tuples take less storage space than lists for the same elements
tuple_ex = (1, 2, 'Obama')
list_ex = [1, 2, 'Obama']
print("Space taken by tuple =",tuple_ex.__sizeof__()," bytes")
print("Space taken by list =",list_ex.__sizeof__()," bytes")
```

```
Space taken by tuple = 48  bytes
Space taken by list = 64  bytes
```

```
#Examples showing that a tuples are not copied, while lists can be copied
tuple_copy = tuple(tuple_ex)
print("Is tuple_copy same as tuple_ex?", tuple_ex is tuple_copy)
list_copy = list(list_ex)
print("Is list_copy same as list_ex?",list_ex is list_copy)
```

```
Is tuple_copy same as tuple_ex? True
Is list_copy same as list_ex? False
```

```
#Examples showing tuples takes lesser time to retrieve elements
import time as tm
tt = tm.time()
list_ex = list(range(1000000)) #List containinig whole numbers upto 1 million
a=(list_ex[::-2])
print("Time take to retrieve every 2nd element from a list = ", tm.time()-tt)

tt = tm.time()
tuple_ex = tuple(range(1000000)) #tuple containinig whole numbers upto 1 million
a=(tuple_ex[::-2])
print("Time take to retrieve every 2nd element from a tuple = ", tm.time()-tt)
```

```
Time take to retrieve every 2nd element from a list =  0.03579902648925781
Time take to retrieve every 2nd element from a tuple =  0.02684164047241211
```

2.3 Dictionary

Unlike lists and tuples, a dictionary is an unordered collection of items. Each item stored in a dictionary has a key and value. You can use a key to retrieve the corresponding value from the dictionary. Dictionaries have the type `dict`.

Dictionaries are often used to store many pieces of information e.g. details about a person, in a single variable. Dictionaries are created by enclosing key-value pairs within braces or curly brackets `{` and `}`, colons to separate keys and values, and commas to separate elements of a dictionary.

The dictionary keys and values are python objects. While values can be any python object, keys need to be immutable python objects, like strings, integers, tuples, etc. Thus, a list can be a value, but not a key, as elements of list can be changed.

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping'}
```

Elements of a dictionary can be retrieved by using the corresponding key.

```
dict_example['India']
```

```
'Narendra Modi'
```

2.3.1 Viewing keys and values

```
dict_example.keys()
```

```
dict_keys(['USA', 'India', 'China'])
```

```
dict_example.values()
```

```
dict_values(['Joe Biden', 'Narendra Modi', 'Xi Jinping'])
```

```
dict_example.items()
```

```
dict_items([('USA', 'Joe Biden'), ('India', 'Narendra Modi'), ('China', 'Xi Jinping')])
```

The results of `keys`, `values`, and `items` look like lists. However, they don't support the indexing operator `[]` for retrieving elements.

```
dict_example.items()[1]
```

```
TypeError: 'dict_items' object is not subscriptable
```

2.3.2 Adding and removing elements in a dictionary

New elements can be added to a dictionary by defining a key in square brackets and assiging it to a value:

```
dict_example['Japan'] = 'Fumio Kishida'
dict_example['Countries'] = 4
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida',
 'Countries': 4}
```

Elements can be removed from the dictionary using the `del` method or the `pop` method:

```
#Removing the element having key as 'Countries'
del dict_example['Countries']
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida'}
```

```
#Removing the element having key as 'USA'
dict_example.pop('USA')
```

```
'Joe Biden'
```

```
dict_example
```

```
{'India': 'Narendra Modi', 'China': 'Xi Jinping', 'Japan': 'Fumio Kishida'}
```

New elements can be added, and values of exisiting keys can be changed using the `update` method:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping','Countrie
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 3}
```

```
dict_example.update({'Countries':4, 'Japan':'Fumio Kishida'})
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 4,
 'Japan': 'Fumio Kishida'}
```

2.3.3 Iterating over elements of a dictionary

The items() attribute of a dictionary can be used to iterate over elements of a dictionary.

```
for key,value in dict_example.items():
    print("The Head of State of",key,"is",value)
```

```
The Head of State of USA is Joe Biden
The Head of State of India is Narendra Modi
The Head of State of China is Xi Jinping
The Head of State of Countries is 4
The Head of State of Japan is Fumio Kishida
```

2.3.4 Practice exercise 4

The GDP per capita of USA for most years from 1960 to 2021 is given by the dictionary `D` given in the code cell below.

Find:

1. The GDP per capita in 2015
2. The GDP per capita of 2014 is missing. Update the dictionary to include the GDP per capita of 2014 as the average of the GDP per capita of 2013 and 2015.
3. Impute the GDP per capita of other missing years in the same manner as in (2), i.e., as the average GDP per capita of the previous year and the next year. Note that the GDP per capita is not missing for any two consecutive years.
4. Print the years and the imputed GDP per capita for the years having a missing value of GDP per capita in (3).

```
D = {'1960':3007,'1961':3067,'1962':3244,'1963':3375,'1964':3574,'1965':3828,'1966':4146,'
```

Solution:

```
print("GDP per capita in 2015 =", D['2015'])
D['2014'] = (D['2013']+D['2015'])/2
for i in range(1960,2021):
    if str(i) not in D.keys():
        D[str(i)] = (D[str(i-1)]+D[str(i+1)])/2
        print("Imputed GDP per capita for the year",i,"is $",D[str(i)])
```

```
GDP per capita in 2015 = 56763
Imputed GDP per capita for the year 1969 is $ 4965.0
Imputed GDP per capita for the year 1977 is $ 9578.5
Imputed GDP per capita for the year 1999 is $ 34592.0
```

2.4 Functions

If an algorithm or block of code is being used several times in a code, then it can be separately defined as a function. This makes the code more organized and readable. For example, let us define a function that prints prime numbers between `a` and `b`, and returns the number of prime numbers found.

```
#Function definition
def prime_numbers (a,b=100):
```

```
    num_prime_nos = 0

    #Iterating over all numbers between a and b
    for i in range(a,b):
        num_divisors=0

        #Checking if the ith number has any factors
        for j in range(2, i):
            if i%j == 0:
                num_divisors=1;break;

        #If there are no factors, then printing and counting the number as prime
        if num_divisors==0:
            print(i)
            num_prime_nos = num_prime_nos+1

    #Return count of the number of prime numbers
    return num_prime_nos
```

In the above function, the keyword `def` is used to define the function, `prime_numbers` is the name of the function, `a` and `b` are the arguments that the function uses to compute the output.

Let us use the defined function to print and count the prime numbers between 40 and 60.

```
#Printing prime numbers between 40 and 60
num_prime_nos_found = prime_numbers(40,60)
```

```
41
43
47
53
59
```

```
num_prime_nos_found
```

```
5
```

If the user calls the function without specifying the value of the argument `b`, then it will take the default value of `100`, as mentioned in the function definition. However, for the argument `a`, the user will need to specify a value, as there is no value defined as a default value in the function definition.

2.4.1 Global and local variables with respect to a function

A variable defined within a function is local to that function, while a variable defined outside the function is global to that function. In case a variable with the same name is defined both outside and inside a function, it will refer to its global value outside the function and local value within the function.

The example below shows a variable with the name `var` referring to its local value when called within the function, and global value when called outside the function.

```
var = 5
def sample_function(var):
    print("Local value of 'var' within 'sample_function()'= ",var)

sample_function(4)
print("Global value of 'var' outside 'sample_function()' = ",var)
```

```
Local value of 'var' within 'sample_function()'=  4
Global value of 'var' outside 'sample_function()' =  5
```

2.4.2 Practice exercise 5

The object `deck` defined below corresponds to a deck of cards. Estimate the probablity that a five card hand will be a flush, as follows:

1. Write a function that accepts a hand of 5 cards as argument, and returns whether the hand is a flush or not.
2. Randomly pull a hand of 5 cards from the deck. Call the function developed in (1) to determine if the hand is a flush.
3. Repeat (2) 10,000 times.
4. Estimate the probability of the hand being a flush from the results of the 10,000 simulations.

You may use the function shuffle() from the `random` library to shuffle the deck everytime before pulling a hand of 5 cards.

```
deck = [{'value':i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
for i in range(2,15)]
```

Solution:

```
import random as rm

#Function to check if a 5-card hand is a flush
def chck_flush(hands):

    #Assuming that the hand is a flush, before checking the cards
    yes_flush =1

    #Storing the suit of the first card in 'first_suit'
    first_suit = hands[0]['suit']

    #Iterating over the remaining 4 cards of the hand
    for j in range(1,len(hands)):

        #If the suit of any of the cards does not match the suit of the first card, the ha
        if first_suit!=hands[j]['suit']:
            yes_flush = 0;

            #As soon as a card with a different suit is found, the hand is not a flush and
            break;
    return yes_flush

flush=0
for i in range(10000):

    #Shuffling the deck
    rm.shuffle(deck)

    #Picking out the first 5 cards of the deck as a hand and checking if they are a flush
    #If the hand is a flush it is counted
    flush=flush+chck_flush(deck[0:5])

print("Probability of obtaining a flush=", 100*(flush/10000),"%")
```

```
Probability of obtaining a flush= 0.18 %
```

2.5 Practice exercise 6

The code cell below defines an object having the nutrition information of drinks in starbucks. Assume that the manner in which the information is structured is consistent throughout the

object.

```
starbucks_drinks_nutrition={'Cool Lime Starbucks Refreshers Beverage': [{'Nutrition_type': 'Calo
, 'value': 1}, {'Nutrition_type': 'Sodium', 'value': 10}], 'Featured Dark Roast
```

Use the object above to answer the following questions:

2.5.1

What is the datatype of the object?

```
print("Datatype=",type(starbucks_drinks_nutrition))
```

```
Datatype= <class 'dict'>
```

2.5.1.1

If the object in (1) is a dictonary, what is the datatype of the values of the dictionary?

```
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks_drinks_nutrition.keys())[
```

```
Datatype= <class 'list'>
```

2.5.1.2

If the object in (1) is a dictonary, what is the datatype of the elements within the values of the dictionary?

```
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks_drinks_nutrition.keys())[
```

```
Datatype= <class 'dict'>
```

2.5.1.3

How many calories are there in `Iced Coffee`?

```
print("Calories = ",starbucks_drinks_nutrition['Iced Coffee'][0]['value'])
```

```
Calories =  5
```

2.5.1.4

Which drink(s) have the highest amount of protein in them, and what is that protein amount?

```
#Defining an empty dictionary that will be used to store the protein of each drink
protein={}

for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Protein':
            protein[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair having the maximum value in the
{key:value for key, value in protein.items() if value == max(protein.values())}
```

```
{'Starbucks® Doubleshot Protein Dark Chocolate': 20,
 'Starbucks® Doubleshot Protein Vanilla': 20,
 'Chocolate Smoothie': 20}
```

2.5.1.5

Which drink(s) have a fat content of more than 10g, and what is their fat content?

```
#Defining an empty dictionary that will be used to store the fat of each drink
fat={}
for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Fat':
            fat[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair having the value more than 10
{key:value for key, value in fat.items() if value>=10}
```

```
{'Starbucks® Signature Hot Chocolate': 26.0, 'White Chocolate Mocha': 11.0}
```

2.5.1.6

Answer Q2.5.1.5 using only dictionary comprehension.

Part II

Exploratory data analysis

3 Reading Data

3.1 3.1 Types of data

In this course, we will focus on analyzing structured data.

3.2 3.2 Using Pandas to Read CSVs

Pandas is a popular Python library used for working in tabular data (similar to the data stored in a spreadsheet). Pandas provides helper functions to read data from various file formats like CSV, Excel spreadsheets, HTML tables, JSON, SQL, and more.

The below format of storing data is known as *comma-separated values* or CSV. It contains day-wise Covid-19 data for Italy:

```
date,new_cases,new_deaths,new_tests
2020-04-21,2256.0,454.0,28095.0
2020-04-22,2729.0,534.0,44248.0
2020-04-23,3370.0,437.0,37083.0
2020-04-24,2646.0,464.0,95273.0
2020-04-25,3021.0,420.0,38676.0
2020-04-26,2357.0,415.0,24113.0
2020-04-27,2324.0,260.0,26678.0
2020-04-28,1739.0,333.0,37554.0
...
```

> **CSVs**: A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. (Wikipedia)

First, let's import the Pandas library. As a convention, it is imported with the alias `pd`.

```
import pandas as pd
import os
```

3.2.1 3.2.1 Using the `read_csv` function

The `pd.read_csv` function can be used to read a CSV file into a pandas `DataFrame`: a spreadsheet-like object for analyzing and processing data.

```
movie_ratings = pd.read_csv('../data/movie_ratings.csv')
```

The built-in python function `type` can be used to check the dataype of an object:

```
type(movie_ratings)
```

```
pandas.core.frame.DataFrame
```

We'll learn more about `DataFrame` in a future lesson.

Note that I use the `relative path` to specify the file path for *movie_ratings.csv*, you may need to change it based on where you store the data file.

3.2.2 3.2.2 Data Overview

Once the data has been read, we may want to see what the data looks like. We'll use another Pandas function `head()` to view the first few rows of the data.

```
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	PG/PG-13
1	Major Dundee	14873	14873	3800000	Apr 07 1965	PG/PG-13
2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	PG/PG-13

Row Indices and column names (axis labels)

By default, when you create a pandas DataFrame (or Series) without specifying an index, pandas will automatically assign integer-based row indices starting from 0. These indices

serve as the row labels and uniquely identify each row in the DataFrame. For example, the index 2 correponds to the row of the movie The Informers. By default, the indices are integers starting from 0. However, they can be changed (to even non-integer values) if desired by the user.

The bold text on top of the DataFrame refers to column names. For example, the column `US Gross` consists of the gross revenue of a movie in the US.

Collectively, the indices and column names are referred as **axis labels**.

Basic information We can view some basic information about the data frame using the `.info` method.

```
movie_ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2228 entries, 0 to 2227
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Title              2228 non-null   object
 1   US Gross           2228 non-null   int64
 2   Worldwide Gross    2228 non-null   int64
 3   Production Budget  2228 non-null   int64
 4   Release Date       2228 non-null   object
 5   MPAA Rating        2228 non-null   object
 6   Source             2228 non-null   object
 7   Major Genre        2228 non-null   object
 8   Creative Type      2228 non-null   object
 9   IMDB Rating        2228 non-null   float64
 10  IMDB Votes         2228 non-null   int64
dtypes: float64(1), int64(4), object(6)
memory usage: 191.6+ KB
```

The `shape` property of a pandas DataFrame provides a tuple that represents the dimensions of the DataFrame:

- The first value in the tuple is the number of rows.
- The second value in the tuple is the number of columns.

```
movie_ratings.shape
```

```
(2228, 11)
```

The `columns` property contains the list of columns within the data frame.

```
movie_ratings.columns
```

```
Index(['Title', 'US Gross', 'Worldwide Gross', 'Production Budget',
       'Release Date', 'MPAA Rating', 'Source', 'Major Genre', 'Creative Type',
       'IMDB Rating', 'IMDB Votes'],
      dtype='object')
```

You can view statistical information for numerical columns (mean, standard deviation, minimum/maximum values, and the number of non-empty values) using the `.describe` method.

```
movie_ratings.describe()
```

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes
count	2.228000e+03	2.228000e+03	2.228000e+03	2228.000000	2228.000000
mean	5.076370e+07	1.019370e+08	3.816055e+07	6.239004	33585.154847
std	6.643081e+07	1.648589e+08	3.782604e+07	1.243285	47325.651561
min	0.000000e+00	8.840000e+02	2.180000e+02	1.400000	18.000000
25%	9.646188e+06	1.320737e+07	1.200000e+07	5.500000	6659.250000
50%	2.838649e+07	4.266892e+07	2.600000e+07	6.400000	18169.000000
75%	6.453140e+07	1.200000e+08	5.300000e+07	7.100000	40092.750000
max	7.601676e+08	2.767891e+09	3.000000e+08	9.200000	519541.000000

Functions & Methods we've looked so far

- `pd.read_csv` - Read data from a CSV file into a Pandas `DataFrame` object
- `.info()` - View basic infomation about rows, columns & data types
- `.shape` - Get the number of rows & columns as a tuple
- `.columns` - Get the list of column names
- `.describe()` - View statistical information about numeric columns

3.3 3.3 Data Selection and Filtering

3.3.1 3.3.1 Extracting Column(s) from pandas

The first step when working with a DataFrame is often to extract one or more columns. To do this effectively, it's helpful to understand the internal structure of a DataFrame. Conceptually,

you can think of a DataFrame as a dictionary of lists, where the keys are column names, and the values are lists or arrays containing data for the respective columns.

```
# Pandas format is simliar to this
movie_ratings_dict = {
    'Title':  ['Opal Dreams', 'Major Dundee', 'The Informers', 'Buffalo Soldiers', 'The La
    'US Gross':  [14443, 14873, 315000, 353743, 388390],
    'Worldwide Gross': [14443, 14873, 315000, 353743, 388390],
    'Production Budget': [9000000, 3800000, 18000000, 15000000, 2200000]
}
```

For dictionary, we use key to retrive its values

```
movie_ratings_dict['Title']
```

```
['Opal Dreams',
 'Major Dundee',
 'The Informers',
 'Buffalo Soldiers',
 'The Last Sin Eater']
```

Similar like dictionary, we can extract a column by its column name

```
movie_ratings['Title']
```

```
0                         Opal Dreams
1                        Major Dundee
2                       The Informers
3                     Buffalo Soldiers
4                   The Last Sin Eater
                     ...
2223                      King Arthur
2224                            Mulan
2225                       Robin Hood
2226    Robin Hood: Prince of Thieves
2227                       Spiceworld
Name: Title, Length: 2228, dtype: object
```

Each column is a feature of the dataframe, we can also use. operator to extract a single column

```
movie_ratings.Title
```

```
0                         Opal Dreams
1                        Major Dundee
2                       The Informers
3                     Buffalo Soldiers
4                   The Last Sin Eater
                     ...
2223                       King Arthur
2224                             Mulan
2225                        Robin Hood
2226    Robin Hood: Prince of Thieves
2227                        Spiceworld
Name: Title, Length: 2228, dtype: object
```

When extracting multiple columns, you need to place the column names inside a list.

```
movie_ratings[['Title', 'US Gross', 'Worldwide Gross' ]]
```

	Title	US Gross	Worldwide Gross
0	Opal Dreams	14443	14443
1	Major Dundee	14873	14873
2	The Informers	315000	315000
3	Buffalo Soldiers	353743	353743
4	The Last Sin Eater	388390	388390
...	...	...	...
2223	King Arthur	51877963	203877963
2224	Mulan	120620254	303500000
2225	Robin Hood	105269730	310885538
2226	Robin Hood: Prince of Thieves	165493908	390500000
2227	Spiceworld	29342592	56042592

3.3.2 3.3.2 Creating new columns from existing columns

New variables (or columns) can be created based on existing variables, or with external data (we'll see adding external data later). For example, let us create a new variable `ratio_wgross_by_budget`, which is the ratio of `Worldwide Gross` and `Production Budget` for each movie:

```
movie_ratings['ratio_wgross_by_budget'] = movie_ratings['Worldwide Gross']/movie_ratings['
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	PG/PG-13
1	Major Dundee	14873	14873	3800000	Apr 07 1965	PG/PG-13
2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	PG/PG-13

3.3.3 3.3.3 Extracting a sub-set of data: `loc` and `iloc`

Sometimes we may be interested in working with a subset of rows and columns of the data, instead of working with the entire dataset. The indexing operators loc and iloc provide a convenient way of selecting a subset of desired rows and columns.

Let us first sort the `movie_ratings` data frame by `IMDB Rating`.

```
movie_ratings_sorted = movie_ratings.sort_values(by = 'IMDB Rating', ascending = False)
movie_ratings_sorted.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MP
182	The Shawshank Redemption	28241469	28241469	25000000	Sep 23 1994	R
2084	Inception	285630280	753830280	160000000	Jul 16 2010	PG
2092	Toy Story 3	410640665	1046340665	200000000	Jun 18 2010	G
1962	Pulp Fiction	107928762	212928762	8000000	Oct 14 1994	R
790	Schindler's List	96067179	321200000	25000000	Dec 15 1993	R

3.3.3.1 3.3.3.1 Subsetting the DataFrame by `loc`

The operator `loc` uses axis labels (row indices and column names) to subset the data.

Let's subset the `title`, `worldwide gross`, `production budget`, and `IMDB raring` of top 3 movies.

```
# Subsetting the DataFrame by loc - using axis labels
movies_subset = movie_ratings_sorted.loc[[182,2084, 2092],[ 'Title', 'IMDB Rating', 'US Gr
movies_subset
```

	Title	IMDB Rating	US Gross	Worldwide Gross	Production Budget
182	The Shawshank Redemption	9.2	28241469	28241469	25000000
2084	Inception	9.1	285630280	753830280	160000000
2092	Toy Story 3	8.9	410640665	1046340665	200000000

The `:` symbol in `.loc` and `.iloc` is a slicing operator that represents a range or all elements in the specified dimension (rows or columns). Use `:` alone to select all rows/columns, or with start/end points to slice specific parts of the DataFrame.

```
# Subsetting the DataFrame by loc - using axis labels. the colon is used to select all row
movies_subset = movie_ratings_sorted.loc[:,['Title','Worldwide Gross','Production Budget',
movies_subset
```

	Title	Worldwide Gross	Production Budget	IMDB Rating
182	The Shawshank Redemption	28241469	25000000	9.2
2084	Inception	753830280	160000000	9.1
2092	Toy Story 3	1046340665	200000000	8.9
1962	Pulp Fiction	212928762	8000000	8.9
790	Schindler's List	321200000	25000000	8.9
...	...	...	...	...
516	Son of the Mask	59918422	100000000	2.0
1495	Disaster Movie	34690901	20000000	1.7
1116	Crossover	7009668	5600000	1.7
805	From Justin to Kelly	4922166	12000000	1.6
1147	Super Babies: Baby Geniuses 2	9109322	20000000	1.4

```
# Subsetting the DataFrame by loc - using axis labels. the colon is used to select a range
movies_subset = movie_ratings_sorted.loc[182:561,['Title','Worldwide Gross','Production Bu
movies_subset
```

	Title	Worldwide Gross	Production Budget	IMDB Rating
182	The Shawshank Redemption	28241469	25000000	9.2
2084	Inception	753830280	160000000	9.1
2092	Toy Story 3	1046340665	200000000	8.9
1962	Pulp Fiction	212928762	8000000	8.9
790	Schindler's List	321200000	25000000	8.9
561	The Dark Knight	1022345358	185000000	8.9

3.3.3.2 3.3.3.2 Subsetting the DataFrame by `iloc`

while `iloc` uses the position of rows or columns, where position has values 0,1,2,3,...and so on, for rows from top to bottom and columns from left to right. In other words, the first row has position 0, the second row has position 1, the third row has position 2, and so on. Similarly, the first column from left has position 0, the second column from left has position 1, the third column from left has position 2, and so on.

```
movie_ratings_sorted.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MP
182	The Shawshank Redemption	28241469	28241469	25000000	Sep 23 1994	R
2084	Inception	285630280	753830280	160000000	Jul 16 2010	PG
2092	Toy Story 3	410640665	1046340665	200000000	Jun 18 2010	G
1962	Pulp Fiction	107928762	212928762	8000000	Oct 14 1994	R
790	Schindler's List	96067179	321200000	25000000	Dec 15 1993	R

```
movie_ratings_sorted.iloc[0:3,[0,2,3,9]]
```

	Title	Worldwide Gross	Production Budget	IMDB Rating
182	The Shawshank Redemption	28241469	25000000	9.2
2084	Inception	753830280	160000000	9.1
2092	Toy Story 3	1046340665	200000000	8.9

```
# Subsetting the DataFrame by iloc - using index of the position of rows and columns
movies_iloc_subset = movie_ratings_sorted.iloc[182:561,[0,2,3,9]]
movies_iloc_subset
```

	Title	Worldwide Gross	Production Budget	IMDB Rating
227	The Boy in the Striped Pyjamas	39830581	12500000	7.8
1463	Lage Raho Munnabhai	31517561	2700000	7.8
363	Coraline	124062750	60000000	7.8
1628	Lucky Number Slevin	55495466	27000000	7.8
1418	Dark City	27257061	27000000	7.8
...	...	...	...	...
1720	Coach Carter	76669806	45000000	7.1
249	Little Women	50003303	15000000	7.1

	Title	Worldwide Gross	Production Budget	IMDB Rating
1752	Drag Me To Hell	85724728	30000000	7.1
1150	Black Snake Moan	9396870	15000000	7.1
665	Find Me Guilty	1788077	13000000	7.1

Why `iloc` returns different rows?

```
# Subsetting the DataFrame by iloc - using index of the position of rows and columns
movies_iloc_subset1 = movie_ratings_sorted.iloc[0:10,[0,2,3,9]]
movies_iloc_subset1
```

	Title	Worldwide Gross	Production Budget	IMDB R
182	The Shawshank Redemption	28241469	25000000	9.2
2084	Inception	753830280	160000000	9.1
2092	Toy Story 3	1046340665	200000000	8.9
1962	Pulp Fiction	212928762	8000000	8.9
790	Schindler's List	321200000	25000000	8.9
561	The Dark Knight	1022345358	185000000	8.9
184	Cidade de Deus	28763397	3300000	8.8
487	The Lord of the Rings: The Fellowship of the Ring	868621686	109000000	8.8
497	The Lord of the Rings: The Return of the King	1133027325	94000000	8.8
1081	C'era una volta il West	5321508	5000000	8.8

3.3.3.3 3.3.3.3 Key differences between`loc` and `iloc` in pandas

- **Indexing Type:**
  - loc uses labels (names) for indexing.
  - iloc uses integer positions for indexing.
- **Inclusion of Endpoints:**
  - In a loc slice, both endpoints are included.
  - In an iloc slice, the endpoint is excluded.

Example:

```
# Assuming you have a DataFrame like this:
import pandas as pd
```

```
data = {'A': [1, 2, 3, 4, 5],
        'B': [10, 20, 30, 40, 50],
        'C': [100, 200, 300, 400, 500]}

df = pd.DataFrame(data, index=['row1', 'row2', 'row3', 'row4', 'row5'])
df
```

	A	B	C
row1	1	10	100
row2	2	20	200
row3	3	30	300
row4	4	40	400
row5	5	50	500

```
# using 'loc'
df.loc['row2':'row4', 'B']
```

```
row2    20
row3    30
row4    40
Name: B, dtype: int64
```

```
# using 'iloc'
df.iloc[1:4, 1]
```

```
row2    20
row3    30
row4    40
Name: B, dtype: int64
```

Note that in the `loc` example, both 'row2' and 'row4' are included in the result, whereas in the `iloc` example, the row at position 4 is excluded.

3.3.4 3.3.4 Extracting rows based on a Single Condition or Multiple Conditions

In many cases, we need to filter data based on specific conditions or a combination of multiple conditions. Next, let's explore how to use these conditions effectively to extract rows that meet our criteria, whether it's a single condition or multiple conditions combined

```
# extracting the rows that have IMDB Rating greater than 8
movie_ratings[movie_ratings['IMDB Rating'] > 8]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Ra
21	Gandhi, My Father	240425	1375194	5000000	Aug 03 2007	Other
56	Ed Wood	5828466	5828466	18000000	Sep 30 1994	R
67	Requiem for a Dream	3635482	7390108	4500000	Oct 06 2000	Other
164	Trainspotting	16501785	24000785	3100000	Jul 19 1996	R
181	The Wizard of Oz	28202232	28202232	2777000	Aug 25 2039	G
...	...	...	...	...	...	...
2090	Finding Nemo	339714978	867894287	94000000	May 30 2003	G
2092	Toy Story 3	410640665	1046340665	200000000	Jun 18 2010	G
2094	Avatar	760167650	2767891499	237000000	Dec 18 2009	PG/PG-1
2130	Scarface	44942821	44942821	25000000	Dec 09 1983	Other
2194	The Departed	133311000	290539042	90000000	Oct 06 2006	R

To combine multiple conditions in pandas, you need to use the `&` (AND) and `|` (OR) operators. Make sure to enclose each condition in parentheses () for clarity and to ensure proper evaluation order.

```
# extracting the rows that have IMDB Rating greater than 8 and US Gross less than 1000000
movie_ratings[(movie_ratings['IMDB Rating'] > 8) & (movie_ratings['US Gross'] < 1000000)]
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
21	Gandhi, My Father	240425	1375194	5000000	Aug 03 2007	Other
636	Lake of Fire	25317	25317	6000000	Oct 03 2007	Other

Combining `.loc` with condition(s) to extract specific rows and columns based on criteria

```
# extracting the rows that have IMDB Rating greater than 8 or US Gross less than 1000000,
movie_ratings[(movie_ratings['IMDB Rating'] > 8) & (movie_ratings['US Gross'] < 1000000)][

#using loc to extract the rows that have IMDB Rating greater than 8 or US Gross less than
movie_ratings.loc[(movie_ratings['IMDB Rating'] > 8) & (movie_ratings['US Gross'] < 100000
```

	Title	IMDB Rating
21	Gandhi, My Father	8.1

	Title	IMDB Rating
636	Lake of Fire	8.4

Can you use `.iloc` for conditional filtering, why or why not?

3.3.5 3.3.5 Finding minimum/maximum of a column

When working with pandas, there are two main options for locating the minimum or maximum values in a DataFrame column:

- `idxmin()` and `idxmax()`: return the **index label** of the first occurrence of the maximum or minimum value in a specified column.

```
# movie_ratings_sorted.iloc[position_max_wgross,:]
max_index = movie_ratings_sorted['Worldwide Gross'].idxmax()
min_index = movie_ratings_sorted['Worldwide Gross'].idxmin()
print("Max index: ", max_index)
print("Min index: ", min_index)
```

```
Max index:  2094
Min index:  896
```

`idxmin()` and `idxmax()` return the index label of the minimum or maximum value in a column. You can use these returned index labels with `.loc` to extract the corresponding row.

```
print(movie_ratings_sorted.loc[max_index,'Worldwide Gross'])
print(movie_ratings_sorted.loc[min_index,'Worldwide Gross'])
```

```
2767891499
884
```

- `argmax()` and `argmin()`: Return the **integer position** of the first occurrence of the maximum or minimum value in a column. You can use these integer positions with `.iloc` to extract the corresponding row

```
# using argmax and argmin, which return the index of the maximum and minimum values
max_position = movie_ratings_sorted['Worldwide Gross'].argmax()
min_position = movie_ratings_sorted['Worldwide Gross'].argmin()
print("max position:", max_position)
```

```
print("min position:", min_position)

# using iloc to get the row with the maximum and minimum values
print(movie_ratings_sorted.iloc[max_position, 2])
print(movie_ratings_sorted.iloc[min_position, 2])
```

```
max position: 43
min position: 2146
2767891499
884
```

Additional Tips: * If you are dealing with non-unique or non-default indices, prefer using `idxmax()/idxmin()` to get the index labels, as `argmax()` might be less intuitive in such cases. * For DataFrames, consider using `.idxmax(axis=1)` or `.idxmin(axis=1)` to find the max/min index labels along rows instead of columns.

3.4 3.4 DataType and DataType Conversion

```
movie_ratings.dtypes
```

```
Title                 object
US Gross               int64
Worldwide Gross        int64
Production Budget      int64
Release Date          object
MPAA Rating           object
Source                object
Major Genre           object
Creative Type         object
IMDB Rating          float64
IMDB Votes             int64
dtype: object
```

The `dtypes` property is used to find the dtypes in the DataFrame.

This returns a Series with the data type of each column.

While it's common for columns containing strings to have the `object` data type, it can also include other types such as lists, dictionaries, or even mixed types within the same column. The `object` data type is a catch-all for columns that contain mixed types or types that aren't easily categorized.

3.4.1 3.4.1 Available Data Types and Associated Built-in Functions

In a DataFrame, columns can have different data types. Here are the common data types you'll encounter and some built-in functions associated with each type:

1. **Numerical Data (int, float)**
    - Built-in functions: `mean()`, `sum()`, `min()`, `max()`, `std()`, `median()`, `quantile()`, etc.
2. **Object Data (str or mixed types)**
    - Built-in functions: `str.contains()`, `str.startswith()`, `str.endswith()`, `str.lower()`, `str.upper()`, `str.replace()`, etc.
3. **Datetime Data (datetime64)**
    - Built-in functions: `dt.year`, `dt.month`, `dt.day`, `dt.strftime()`, `dt.weekday()`, `dt.hour`, etc.

These functions help in exploring and transforming the data effectively depending on the type of data in each column.

3.4.2 3.4.2 Data Type Conversion

When you work on a specific column, being mindful of which data type it is, the data type depends on its built in function.

Often, we need to convert the datatypes of some of the columns to make them suitable for analysis. For example, the datatype of Release Date in the DataFrame movie_ratings is object. To perform datetime related computations on this variable, we'll need to convert it to a datatime format. We'll use the Pandas function `to_datatime()` to covert it to a datatime format. Similar functions such as `to_numeric()`, `to_string()` etc., can be used for other conversions.

```
movie_ratings['Release Date']
```

```
0       Nov 22 2006
1       Apr 07 1965
2       Apr 24 2009
3       Jul 25 2003
4       Feb 09 2007
           ...
2223    Jul 07 2004
```

```
2224    Jun 19 1998
2225    May 14 2010
2226    Jun 14 1991
2227    Jan 23 1998
Name: Release Date, Length: 2228, dtype: object
```

```
# check the datatype of release data column
movie_ratings['Release Date'].dtypes
```

```
dtype('O')
```

We can see above that the function `to_datetime()` converts Release Date to a `datetime` format.

Next, we'll update the variable `Release Date` in the DataFrame to be in the `datetime` format:

```
movie_ratings['Release Date'] = pd.to_datetime(movie_ratings['Release Date'])
```

```
# Let's check the datatype of release data column again
movie_ratings['Release Date'].dtypes
```

```
dtype('<M8[ns]')
```

`dtype('<M8[ns]')` means a 64-bit datetime object with nanosecond precision stored in little-endian format. This data type is commonly used to represent timestamps in high-resolution time series data.

Next, we can use the built-in datetime functions to extract the year from this variable and create the 'release year' column.

```
# Extracting the year from the release date
movie_ratings['Release Year'] = movie_ratings['Release Date'].dt.year
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	2006-11-22	PG/PG-13
1	Major Dundee	14873	14873	3800000	1965-04-07	PG/PG-13
2	The Informers	315000	315000	18000000	2009-04-24	R

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
3	Buffalo Soldiers	353743	353743	15000000	2003-07-25	R
4	The Last Sin Eater	388390	388390	2200000	2007-02-09	PG/PG-13

In Pandas, the `errors='coerce'` parameter is often used in the context of data conversion, specifically when using the `pd.to_numeric` function. This argument tells Pandas to convert values that it can and set the ones it cannot convert to `NaN`. It's a way of gracefully handling errors without raising an exception. Read the textbook for an example

3.4.3 3.4.3 Data Type Filtering

We can filter the columns based on its data types

```
# select just object columns
movie_ratings.select_dtypes(include='object').head()
```

	Title	MPAA Rating	Source	Major Genre	Creative Type
0	Opal Dreams	PG/PG-13	Adapted screenplay	Drama	Fiction
1	Major Dundee	PG/PG-13	Adapted screenplay	Western/Musical	Fiction
2	The Informers	R	Adapted screenplay	Horror/Thriller	Fiction
3	Buffalo Soldiers	R	Adapted screenplay	Comedy	Fiction
4	The Last Sin Eater	PG/PG-13	Adapted screenplay	Drama	Fiction

```
# select the numeric columns
movie_ratings.select_dtypes(include='number').head()
```

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes	Release Year	ratio_w
0	14443	14443	9000000	6.5	468	2006	0.00160
1	14873	14873	3800000	6.7	2588	1965	0.00391
2	315000	315000	18000000	5.2	7595	2009	0.01750
3	353743	353743	15000000	6.9	13510	2003	0.02358
4	388390	388390	2200000	5.7	1012	2007	0.17654

3.4.4 3.4.4 Summary statistics across rows/columns in Pandas: Numeric Columns

The Pandas DataFrame class has functions such as `sum()` and `mean()` to compute sum over rows or columns of a DataFrame.

By default, functions like `mean()` and `sum()` compute the statistics for each column (i.e., all rows are aggregated) in the DataFrame. Let us compute the mean of all the numeric columns of the data:

```
movie_ratings.describe()
```

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes
count	2.228000e+03	2.228000e+03	2.228000e+03	2228.000000	2228.000000
mean	5.076370e+07	1.019370e+08	3.816055e+07	6.239004	33585.154847
std	6.643081e+07	1.648589e+08	3.782604e+07	1.243285	47325.651561
min	0.000000e+00	8.840000e+02	2.180000e+02	1.400000	18.000000
25%	9.646188e+06	1.320737e+07	1.200000e+07	5.500000	6659.250000
50%	2.838649e+07	4.266892e+07	2.600000e+07	6.400000	18169.000000
75%	6.453140e+07	1.200000e+08	5.300000e+07	7.100000	40092.750000
max	7.601676e+08	2.767891e+09	3.000000e+08	9.200000	519541.000000

```
# select the numeric columns
movie_ratings.mean(numeric_only=True)
```

```
US Gross                 5.076370e+07
Worldwide Gross          1.019370e+08
Production Budget        3.816055e+07
IMDB Rating              6.239004e+00
IMDB Votes               3.358515e+04
Release Year             2.002005e+03
ratio_wgross_by_budget   1.259483e+01
dtype: float64
```

Using the `axis` parameter:

The `axis` parameter controls whether to compute the statistic across rows or columns: * The argument `axis=0`(deafult) denotes that the mean is taken over all the rows of the DataFrame. * For computing a statistic across column the argument `axis=1` will be used.

If mean over a subset of columns is desired, then those column names can be subset from the data.

For example, let us compute the mean IMDB rating, and mean IMDB votes of all the movies:

```
movie_ratings[['IMDB Rating','IMDB Votes']].mean(axis = 0)
```

```
IMDB Rating        6.239004
IMDB Votes     33585.154847
dtype: float64
```

Pandas sum function

```
data = [[10, 18, 11], [13, 15, 8], [9, 20, 3]]
df = pd.DataFrame(data )
df
```

	0	1	2
0	10	18	11
1	13	15	8
2	9	20	3

```
# By default, the sum method adds values accross rows and returns the sum for each column
df.sum()
```

```
0    32
1    53
2    22
dtype: int64
```

```
# By specifying the column axis (axis='columns'), the sum() method add values accross colu
df.sum(axis = 'columns')
```

```
0    39
1    36
2    32
dtype: int64
```

```
# in python, axis=1 stands for column, while axis=0 stands for rows
df.sum(axis = 1)
```

```
0    39
1    36
2    32
dtype: int64
```

3.5 3.5 Writing data to a `.csv` file

The Pandas function `to_csv` can be used to write (or export) data to a csv. Below is an example.

```
#Exporting the data of the top 250 movies to a csv file
movie_ratings.to_csv('../data/movie_rating_exported.csv')
```

```
# check if the file has been exported
os.listdir('../data')
```

```
['bestseller_books.txt',
 'country-capital-lat-long-population.csv',
 'covid.csv',
 'fifa_data.csv',
 'food_quantity.csv',
 'gas_prices.csv',
 'gdp_lifeExpectancy.csv',
 'LOTR 2.csv',
 'LOTR.csv',
 'movies.csv',
 'movies_cleaned.csv',
 'movie_ratings.csv',
 'movie_ratings.txt',
 'movie_rating_exported.csv',
 'party_nyc.csv',
 'price.csv',
 'question_json_data.json',
 'spotify_data.csv',
 'stocks.csv']
```

3.6 3.6 Reading other data formats - txt, html, json

Although `.csv` is a very popular format for structured data, data is found in several other formats as well. Some of the other data formats are `.txt`, `.html` and `.json`.

3.6.1 3.6.1 Reading `.txt` files

The *txt* format offers some additional flexibility as compared to the *csv* format. In the *csv* format, the delimiter is a comma (or the column values are separated by a comma). However, in a *txt* file, the delimiter can be anything as desired by the user. Let us read the file *movie_ratings.txt*, where the variable values are separated by a tab character.

```
movie_ratings_txt = pd.read_csv('../data/movie_ratings.txt',sep='\t')
movie_ratings_txt.head()
```

	Unnamed: 0	Title	US Gross	Worldwide Gross	Production Budget	Release Date	M
0	0	Opal Dreams	14443	14443	9000000	Nov 22 2006	P
1	1	Major Dundee	14873	14873	3800000	Apr 07 1965	P
2	2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	P

We use the function *read_csv* to read a *txt* file. However, we mention the tab character (r"") as a separator of variable values.

Note that there is no need to remember the argument name - *sep* for specifying the delimiter. You can always refer to the read_csv() documentation to find the relevant argument.

3.6.2 Practice exercise 4

Read the file *bestseller_books.txt*. It contains top 50 best-selling books on amazon from 2009 to 2019. Identify the delimiter without opening the file with Notepad or a text-editing software. How many rows and columns are there in the dataset?

Solution:

```
#The delimiter seems to be ';' based on the output of the above code
bestseller_books = pd.read_csv('../Data/bestseller_books.txt',sep=';')
bestseller_books.head()
```

	Unnamed: 0.1	Unnamed: 0	Name	Author
0	0	0	10-Day Green Smoothie Cleanse	JJ Smith
1	1	1	11/22/63: A Novel	Stephen King
2	2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson
3	3	3	1984 (Signet Classics)	George Orwell
4	4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic

```
#The file read with ';' as the delimited is correct
print("The file has",bestseller_books.shape[0],"rows and",bestseller_books.shape[1],"colum
```

```
The file has 550 rows and 9 columns
```

Alternatively, you can use the argument `sep = None`, and `engine = 'python'`. The default engine is C. However, the 'python' engine has a 'sniffer' tool which may identify the delimiter automatically.

```
bestseller_books = pd.read_csv('../data/bestseller_books.txt',sep=None, engine = 'python')
bestseller_books.head()
```

	Unnamed: 0.1	Unnamed: 0	Name	Author
0	0	0	10-Day Green Smoothie Cleanse	JJ Smith
1	1	1	11/22/63: A Novel	Stephen King
2	2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson
3	3	3	1984 (Signet Classics)	George Orwell
4	4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic

3.6.3 3.6.2 Reading HTML data

The *Pandas* function *read_html* searches for tabular data, i.e., data contained within the <*table*> tags of an html file. Let us read the tables in the GDP per capita page on Wikipedia.

```
#Reading all the tables from the Wikipedia page on GDP per capita
tables = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_pe
```

All the tables will be read and stored in the variable named as *tables*. Let us find the datatype of the variable *tables*.

```
#Finidng datatype of the variable - tables
type(tables)
```

```
list
```

The variable - tables is a list of all the tables read from the HTML data.

```
#Number of tables read from the page
len(tables)
```

```
6
```

The in-built function *len* can be used to find the length of the list - *tables* or the number of tables read from the Wikipedia page. Let us check out the first table.

```
#Checking out the first table. Note that the index of the first table will be 0.
tables[0]
```

	0	1	2
0	>$60,000 $50,000–$60,000 $40,000–$50,000 $30,0...	$20,000–$30,000 $10,000–$20,000 $5,000–$10,000...	$

The above table doesn't seem to be useful. Let us check out the second table.

```
#Checking out the second table. Note that the index of the first table will be 1.
tables[1]
```

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
0	Monaco	—	—	240862	2022	234317	2021
1	Liechtenstein	—	—	187267	2022	169260	2021
2	Luxembourg	131384	2024	128259	2023	133745	2021
3	Bermuda	—	—	123091	2022	112653	2021
4	Ireland	106059	2024	103685	2023	101109	2021
...	...	...	...	...	...	...	...
218	Malawi	481	2024	673	2023	613	2021

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
219	South Sudan	422	2024	1072	2015	400	2021
220	Afghanistan	422	2022	353	2022	373	2021
221	Syria	—	—	421	2021	925	2021
222	Burundi	230	2024	200	2023	311	2021

The above table contains the estimated GDP per capita of all countries. This is the table that is likely to be relevant to a user interested in analyzing GDP per capita of countries. Instead of reading all tables of an HTML file, we can focus the search to tables containing certain relevant keywords. Let us try searching all table containing the word 'Country'.

```
#Reading all the tables from the Wikipedia page on GDP per capita, containing the word 'Co
tables = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_pe
```

The *match* argument can be used to specify the keywords to be present in the table to be read.

```
len(tables)
```

```
1
```

Only one table contains the keyword - 'Country'. Let us check out the table obtained.

```
#Table having the keyword - 'Country' from the HTML page
tables[0]
```

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
0	Monaco	—	—	240862	2022	234317	2021
1	Liechtenstein	—	—	187267	2022	169260	2021
2	Luxembourg	131384	2024	128259	2023	133745	2021
3	Bermuda	—	—	123091	2022	112653	2021
4	Ireland	106059	2024	103685	2023	101109	2021
...	...	...	...	...	...	...	...

	Country/Territory	IMF[4][5]		World Bank[6]		United Nations[7]	
	Country/Territory	Estimate	Year	Estimate	Year	Estimate	Year
218	Malawi	481	2024	673	2023	613	2021
219	South Sudan	422	2024	1072	2015	400	2021
220	Afghanistan	422	2022	353	2022	373	2021
221	Syria	—	—	421	2021	925	2021
222	Burundi	230	2024	200	2023	311	2021

The argument *match* helps with a more focussed search, and helps us discard irrelevant tables.

3.6.4 Practice exercise 5

Read the table(s) consisting of attendance of spectators in FIFA worlds cup from this page. Read only those table(s) that have the word *'attendance'* in them. How many rows and columns are there in the table(s)?

```
dfs = pd.read_html('https://en.wikipedia.org/wiki/FIFA_World_Cup',
                   match='reaching')
print(len(dfs))
data = dfs[0]
print("Number of rows =",data.shape[0], "and number of columns=",data.shape[1])
data.head()
```

```
1
Number of rows = 25 and number of columns= 6
```

	Team	Titles	Runners-up	Third place	
0	Brazil	5 (1958, 1962, 1970, 1994, 2002)	2 (1950 *, 1998)	2 (1938, 1978)	2
1	Germany1	4 (1954, 1974 *, 1990, 2014)	4 (1966, 1982, 1986, 2002)	4 (1934, 1970, 2006 *, 2010)	1
2	Italy	4 (1934 *, 1938, 1982, 2006)	2 (1970, 1994)	1 (1990 *)	1
3	Argentina	3 (1978 *, 1986, 2022)	3 (1930, 1990, 2014)	NaN	N
4	France	2 (1998 *, 2018)	2 (2006, 2022)	2 (1958, 1986)	1

3.6.5 3.6.3 Reading JSON data

JSON stands for JavaScript Object Notation, in which the data is stored and transmitted as plain text. A couple of benefits of the JSON format are:

1. Since the format is text only, JSON data can easily be exchanged between web applications, and used by any programming language.
2. Unlike the *csv* format, JSON supports a hierarchical data structure, and is easier to integrate with APIs.

The JSON format can support a hierachical data structure, as it is built on the following two data structures (*Source: technical documentation*):

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

The *Pandas* function read_json converts a JSON string to a Pandas DataFrame. The function *dumps()* of the *json* library converts a Python object to a JSON string.

Lets read the JSON data on Ted Talks.

```
tedtalks_data = pd.read_json('https://raw.githubusercontent.com/cwkenwaysun/TEDmap/master/
```

```
tedtalks_data.head()
```

	id	speaker	headline	URL	descripti
0	7	David Pogue	Simplicity sells	http://www.ted.com/talks/view/id/7	New Yo
1	6	Craig Venter	Sampling the ocean's DNA	http://www.ted.com/talks/view/id/6	Genomic
2	4	Burt Rutan	The real future of space exploration	http://www.ted.com/talks/view/id/4	In this p
3	3	Ashraf Ghani	How to rebuild a broken state	http://www.ted.com/talks/view/id/3	Ashraf G
4	5	Chris Bangle	Great cars are great art	http://www.ted.com/talks/view/id/5	America

```
[{'question': "What is the data type of values in the last column (named 'rates') of the abo
  'type': 'multiple_choice',
  'answers': [{'answer': 'list', 'correct': True, 'feedback': 'Correct!'},
   {'answer': 'string',
```

```
 'correct': False,
 'feedback': 'Incorrect. Use the type function on the variable to find its datatype.'},
{'answer': 'numeric',
 'correct': False,
 'feedback': 'Incorrect. Use the type function on the variable to find its datatype.'},
{'answer': 'dictionary',
 'correct': False,
 'feedback': 'Incorrect. Use the type function on the variable to find its datatype.'}]}]
```

This JSON data contains nested structures, such as lists and dictionaries, which require a deeper understanding to effectively structure. We will address this in future lectures

3.6.6 Practice exercise 6

Read the movies dataset from here. How many rows and columns are there in the data?

```
movies_data = pd.read_json('https://raw.githubusercontent.com/vega/vega-datasets/master/da
print("Number of rows =",movies_data.shape[0], "and number of columns=",movies_data.shape[
```

```
Number of rows = 3201 and number of columns= 16
```

3.6.7 3.6.4 Reading data from a URL in Python

This process typically involves using the `requests` library, which allows you to send HTTP requests and handle responses easily.

You'll need to install it using `pip`

We'll use the CoinGecko API, which provides cryptocurrency market data. Here's an example of how to retrieve current market data:

```
import requests

# Define the URL of the API
url = 'https://api.coingecko.com/api/v3/coins/markets?vs_currency=usd'

# Send a GET request to the URL
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
```

```
    # Parse the JSON data
    data = response.json()
    print(data)
else:
    print(f"Failed to retrieve data: {response.status_code}")
```

```
[{'id': 'bitcoin', 'symbol': 'btc', 'name': 'Bitcoin', 'image': 'https://coin-images.coingecl
```

```
# Loop through the data and print the name and current price
for coin in data:
    name = coin['name']
    price = coin['current_price']
    print(f"Coin: {name}, Price: ${price}")
```

```
Coin: Bitcoin, Price: $62490
Coin: Ethereum, Price: $2434.1
Coin: Tether, Price: $0.999711
Coin: BNB, Price: $568.59
Coin: Solana, Price: $144.62
Coin: USDC, Price: $0.99982
Coin: XRP, Price: $0.531761
Coin: Lido Staked Ether, Price: $2433.54
Coin: Dogecoin, Price: $0.109138
Coin: TRON, Price: $0.156189
Coin: Toncoin, Price: $5.23
Coin: Cardano, Price: $0.35311
Coin: Avalanche, Price: $26.86
Coin: Shiba Inu, Price: $1.759e-05
Coin: Wrapped stETH, Price: $2870.49
Coin: Wrapped Bitcoin, Price: $62339
Coin: WETH, Price: $2434.35
Coin: Chainlink, Price: $11.22
Coin: Bitcoin Cash, Price: $325.66
Coin: Polkadot, Price: $4.15
Coin: Dai, Price: $0.999811
Coin: Sui, Price: $2.06
Coin: NEAR Protocol, Price: $5.11
Coin: LEO Token, Price: $6.01
Coin: Uniswap, Price: $7.24
Coin: Litecoin, Price: $65.04
```

Coin: Bittensor, Price: $617.07
Coin: Aptos, Price: $8.91
Coin: Pepe, Price: $9.89e-06
Coin: Wrapped eETH, Price: $2554.88
Coin: Artificial Superintelligence Alliance, Price: $1.49
Coin: Internet Computer, Price: $8.13
Coin: Kaspa, Price: $0.137693
Coin: POL (ex-MATIC), Price: $0.376926
Coin: Ethereum Classic, Price: $18.7
Coin: Stellar, Price: $0.091505
Coin: Monero, Price: $144.06
Coin: Stacks, Price: $1.77
Coin: First Digital USD, Price: $0.999501
Coin: dogwifhat, Price: $2.56
Coin: OKB, Price: $41.75
Coin: Ethena USDe, Price: $0.998868
Coin: Immutable, Price: $1.49
Coin: Filecoin, Price: $3.74
Coin: Aave, Price: $146.68
Coin: Cronos, Price: $0.078844
Coin: Optimism, Price: $1.67
Coin: Render, Price: $5.32
Coin: Injective, Price: $20.63
Coin: Arbitrum, Price: $0.55172
Coin: Hedera, Price: $0.052714
Coin: Mantle, Price: $0.594723
Coin: Fantom, Price: $0.680296
Coin: VeChain, Price: $0.02305745
Coin: Cosmos Hub, Price: $4.44
Coin: THORChain, Price: $5.09
Coin: WhiteBIT Coin, Price: $11.61
Coin: The Graph, Price: $0.16643
Coin: Sei, Price: $0.436364
Coin: Bitget Token, Price: $1.075
Coin: Bonk, Price: $2.134e-05
Coin: Binance-Peg WETH, Price: $2434.55
Coin: FLOKI, Price: $0.00013831
Coin: Rocket Pool ETH, Price: $2719.44
Coin: Theta Network, Price: $1.31
Coin: Popcat, Price: $1.28
Coin: Arweave, Price: $19.03
Coin: Maker, Price: $1406.89
Coin: Mantle Staked Ether, Price: $2540.6

Coin: MANTRA, Price: $1.4
Coin: Pyth Network, Price: $0.327718
Coin: Helium, Price: $6.89
Coin: Solv Protocol SolvBTC, Price: $62563
Coin: Celestia, Price: $5.39
Coin: Gate, Price: $8.86
Coin: Jupiter, Price: $0.772541
Coin: Algorand, Price: $0.125351
Coin: Polygon, Price: $0.377139
Coin: Ondo, Price: $0.711239
Coin: Worldcoin, Price: $1.95
Coin: Quant, Price: $67.87
Coin: Lido DAO, Price: $1.079
Coin: KuCoin, Price: $7.95
Coin: JasmyCoin, Price: $0.01934656
Coin: Bitcoin SV, Price: $45.98
Coin: Conflux, Price: $0.198997
Coin: BitTorrent, Price: $9.23941e-07
Coin: Brett, Price: $0.088115
Coin: Core, Price: $0.937112
Coin: Fasttoken, Price: $2.6
Coin: GALA, Price: $0.02149496
Coin: ether.fi Staked ETH, Price: $2424.85
Coin: Wormhole, Price: $0.324805
Coin: Flow, Price: $0.540995
Coin: Notcoin, Price: $0.00801109
Coin: Beam, Price: $0.01552574
Coin: Renzo Restaked ETH, Price: $2482.56
Coin: Ethena, Price: $0.287191
Coin: Klaytn, Price: $0.132756
Coin: Aerodrome Finance, Price: $1.2

A Assignment A

Data structures & reading data

Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the *Code* cells and your answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
3. Use Quarto to print the *.ipynb* file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. The assignment is worth 100 points, and is due on **8th October 2023 at 11:59 pm**.
5. **Five points are properly formatting the assignment**. The breakdown is as follows:

- Must be an HTML file rendered using Quarto (2 pts).
- There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) (1 pt)
- Final answers of each question are written in Markdown cells (1 pt).
- There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)

A.1 List comprehension

USA's GDP per capita from 1960 to 2021 is given by the tuple `T` in the code cell below. The values are arranged in ascending order of the year, i.e., the first value is for 1960, the second value is for 1961, and so on.

```
T = (3007, 3067, 3244, 3375,3574, 3828, 4146, 4336, 4696, 5032,5234,5609,6094,6726,7226,78
```

A.1.1

Use list comprehension to create a list of the gaps between consecutive entries in `T`, i.e, the increase in GDP per capita with respect to the previous year. The list with gaps should look like: [60, 177, ...]. Let the name of this list be `GDP_increase`.

(4 points)

A.1.2

Use `GDP_increase` to find the maximum gap size, i.e, the maximum increase in GDP per capita.

(1 point)

A.1.3

Use list comprehension with `GDP_increase` to find the percentage of gaps that have size greater than $1000.

(3 points)

A.1.4

Use list comprehension with `GDP_increase` to print the list of years in which the GDP per capita increase was more than $2000.

Hint: The `enumerate()` function may help.

(4 points)

A.1.5

Use list comprehension to:

1. Create a list that consists of the difference between the maximum and minimum GDP per capita values for each of the 5 year-periods starting from 1976, i.e., for the periods 1976-1980, 1981-1985, 1986-1990, ..., 2016-2020.
2. Find the five year period in which the difference *(between the maximum and minimum GDP per capita values)* was the least.

(4 + 2 points)

A.2 Nested list-comprehension

Below is the list consisting of the majors / minors of students of the course STAT303-1 Fall 2023. This data is a list of lists, where each sub-list *(smaller list within the outer larger list)* consists of the majors / minors of a student. Most of the students have majors / minors in one or more of these four areas:

1. Math / Statistics / Computer Science
2. Humanities / Communication
3. Social Sciences / Education
4. Physical Sciences / Natural Sciences / Engineering

There are some students having majors / minors in other areas as well.

Use list comprehension for all the questions below.

```
majors_minors = majors_minors = [['Humanities / Communications', 'Math / Statistics / Comp
```

A.2.1

How many students have major / minor in any three of the above mentioned four areas?

(1 point)

A.2.2

How many students have *Math / Statistics / Computer Science* as an area of their major / minor?

Hint: Nested list comprehension

(4 points)

A.2.3

How many students have *Math / Statistics / Computer Science* as the **only area** of their major / minor?

Hint: Nested list comprehension

(5 points)

A.2.4

How many students have *Math / Statistics / Computer Science* and *Social Sciences / Education* as a couple of areas of their major / minor?

Hint: The in-built function `all()` may be useful.

(6 points)

A.3 Dictionary

The code cell below defines an object having the nutrition information of drinks in starbucks.

```
starbucks_drinks_nutrition={'Cool Lime Starbucks Refreshers Beverage': [{'Nutrition_type': 'Calo
, 'value': 1}, {'Nutrition_type': 'Sodium', 'value': 10}], 'Featured Dark Roast': [{'Nutrition_type'
```

A.3.1

What is the nested data structure of the object `starbucks_drinks_nutrition`? An example of a nested data structure can be a list of dictionaries, where the dictionary values are a tuple of dictionaries?

(1 point)

A.3.2

Use dictionary-comprehension to print the name and `carb` content of all drinks that have a `carb` content of more than 50 units.

Hint: It will be a nested dictionary comprehension.

(6 points)

A.4 Ted talks

A.4.1

Read the data on ted talks from 2006 to 2017.

(1 point)

A.4.2

Find the number of talks in the dataset.

(1 point)

A.4.3

Find the `headline`, `speaker` and `year_filmed` of the talk with the highest number of views.

(4 points)

A.4.4

Do the majority of talks have less views than the average number of views for a talk? Justify your answer.

(3 points)

Hint: Print summary statistics for questions (4) and (5).

A.4.5

Do at least 25% of the talks have more views than the average number of views for a talk? Justify your answer.

(3 points)

A.4.6

The last column of the dataset consists of votes obtained by the talk under different categories, such as *Funny, Confusing, Fascinating, etc.* For each category, create a new column in the dataset that contains the votes obtained by the tedtalk in that category. Print the first 5 rows of the updated dataset.

(8 points)

A.4.7

With the data created in (a), find the `headline` of the talk that received the highest number of votes as *Confusing*.

(4 points)

A.4.8

With the data created in (a), find the `headline` and the `year` of the talk that received the highest percentage of votes in the *Fascinating* category.

$$\text{Percentage of } \textit{Fascinating} \text{ votes for a ted talk} = \frac{Number\ of\ votes\ in\ the\ category\ `Fascinating`}{Total\ votes\ in\ all\ categories}$$

(7 points)

A.5 University rankings

A.5.1

Download the data set "univ.txt". Read it with python.

(1 point)

A.5.2

Find summary statistics of the data. Based on the statistics, answer the next four questions.

(1 point)

A.5.3

How many universities are there in the data set?

(1 point)

A.5.4

Estimate the maximum `Tuition and fees` among universities that are in the bottom 25% when ranked by total tuition and fees.

(2 points)

A.5.5

How many universities share the ranking of 220? (If `s` universities share the same rank, say `r`, then the next lower rank is `r+s`, and all the ranks in between `r` and `r+s` are dropped)

(4 points)

A.5.6

Can you find the mean `Tuition and fees` for an undergrad student in the US from the summary statistics? Justify your answer.

(3 points)

A.5.7

Find the average `Tuition and fees` for an undergrad student in the US.

(5 points)

A.6 File formats

Consider the file formats - *csv, JSON, txt.* Mention one advantage and one disadvantage of each format over the other two formats.

(2+2+2 = 6 points)