Department of Computer Science
National Textile University Faisalabad

Data Science – Lab Manual

Lab Plan - 1: Crash Course in Python

Outline:

- Introduction to Python
- Variables and Types
- Data Structures in Python
- Functions & Packages
- Numpy package

1.1. Introduction to Python

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming.It was created by Guido van Rossum during 1985- 1990. Python is very beginner-friendly. The syntax (words and structure) is extremely simple to read and follow, most of which can be understood even if you do not know any programming. Let's take a look at one example of this

Example 1.1:

```
Garage = "Ferrari", "Honda", "Porsche", "Toyota"

for each_car in Garage:
     print(each_car)
```

"print()" is a built-in Python function that will output some text to the `console`.

Looking at the code about cars in the garage, can you guess what will happen? You probably have a general idea. For each_car in the garage, we're going to do something. What are we doing? We are printing each car.

Since "printing" outputs some text to the "console," you can probably figure out that the console will say something like "Ferrari, Honda, Porsche, Toyota."

1.1.1. Python Shell

A **python shell** is a way for a user to interact with the **python** interpreter.

1.1.2. Python IDLE

IDLE (Integrated DeveLopment Environment or Integrated Development and Learning Environment) is an integrated development environment for **Python**, which has been bundled with the default implementation of the language since 1.5.2b1. ...**Python** shell with syntax highlighting.

1.1.3. Python Scripts

Scripts are reusable

Basically, a script is a text file containing the statements that comprise a Python program. Once you have created the script, you can execute it over and over without having to retype it each time. Python scripts are saved with .py extension.

Scripts are editable

Perhaps, more importantly, you can make different versions of the script by modifying the statements from one file to the next using a text editor. Then you can execute each of the individual versions. In this way, it is easy to create different programs with a minimum amount of typing.

1.1.4. Anaconda Distribution

The open-source Anaconda Distribution is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over 11 million users worldwide, it is the industry standard for developing, testing, and training on a single machine.

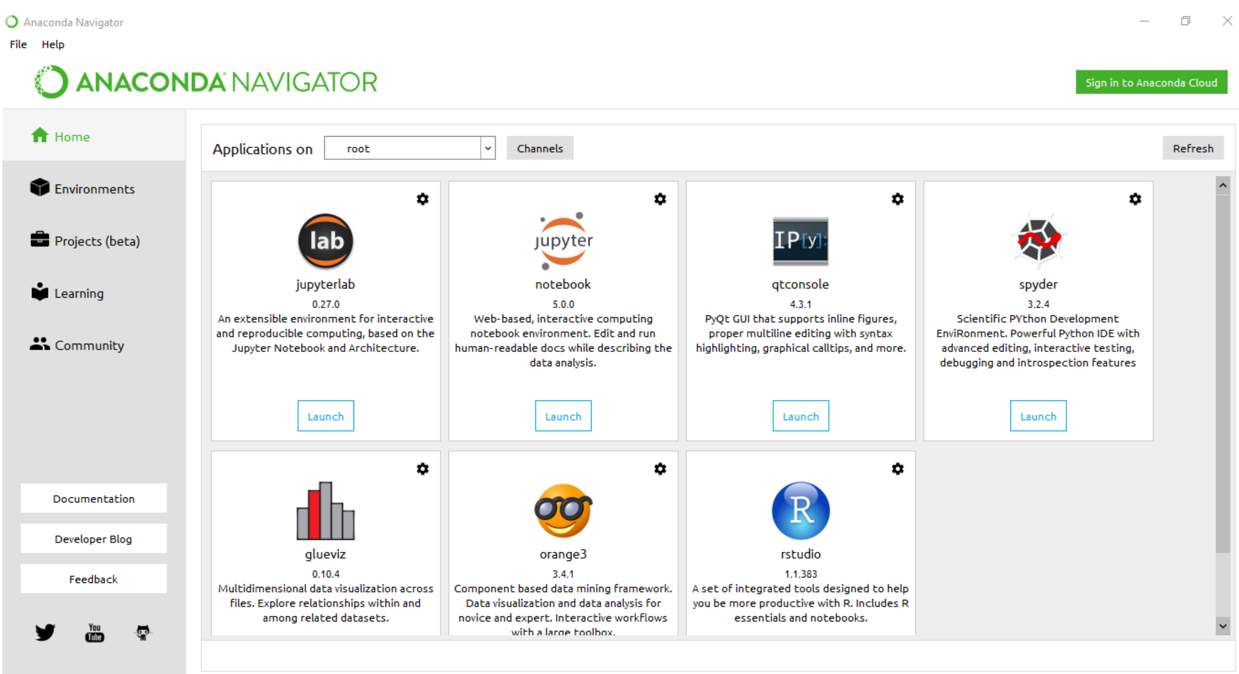

Figure 1.1: Anaconda Navigator

Spyder IDE

Spyder is a powerful scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts. It offers a unique combination of the advanced editing, analysis, debugging, and profiling functionality of a comprehensive development tool with the data exploration, interactive execution, deep inspection, and beautiful visualization capabilities of a scientific package.

Beyond its many built-in features, its abilities can be extended even further via its plugin system and API. Furthermore, Spyder can also be used as a PyQt5 extension library, allowing developers to build upon its functionality and embed its components, such as the interactive console, in their own PyQt software.

Some of the components of spyder include:

- ***Editor***, work efficiently in a multi-language editor with a function/class browser, code analysis tools, automatic code completion
- ***Ipython console,*** harness the power of as many IPython consoles as you like within the flexibility of a full GUI interface; run your code by line, cell, or file; and render plots right inline.
- ***Variable explorer,***Interact with and modify variables on the fly: plot a histogram or time series, edit a data frame or Numpy array, sort a collection, dig into nested objects, and more!
- ***Debugger,***Trace each step of your code's execution interactively.
- ***Help,***instantly view any object's docs, and render your own.

Here is how Spyder IDE looks like (Windows edition) in action:

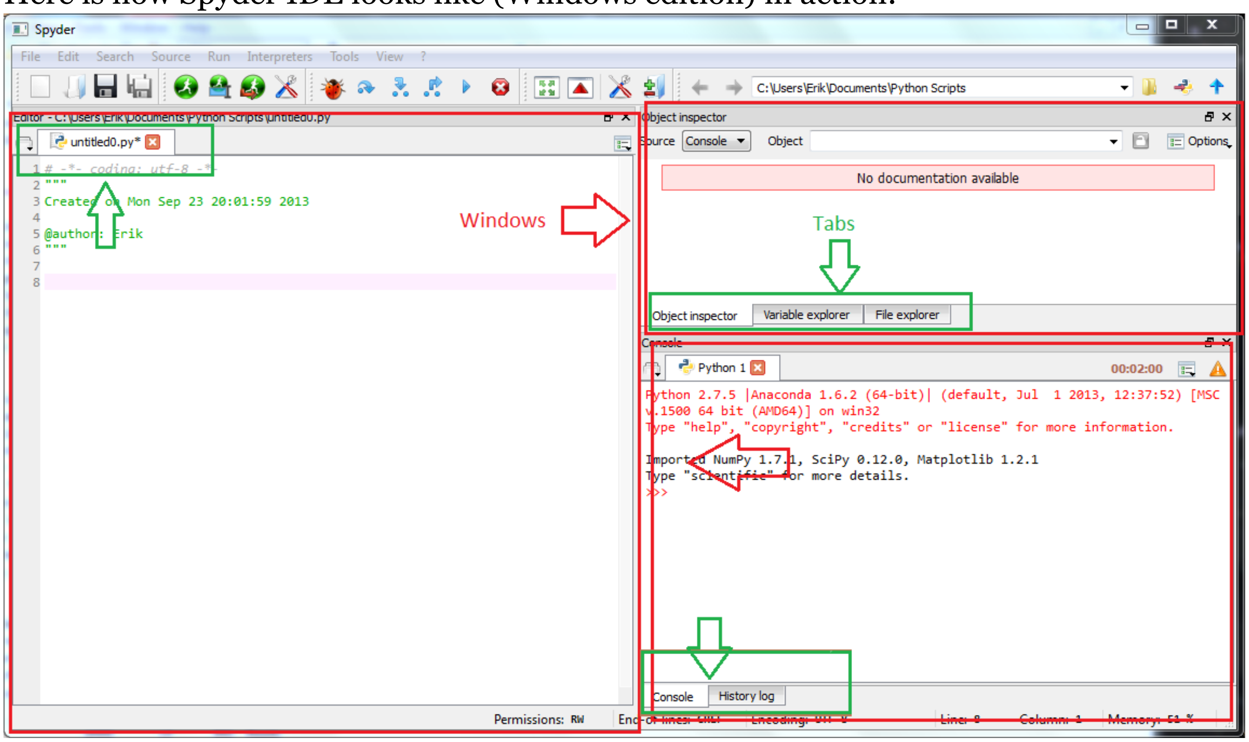

Figure 1.2: Spyder IDE

The first thing to note is how the Spyder app is organized. The application includes multiple separate windows (marked with red rectangles), each of which has its own tabs (marked with green rectangles). You can change which windows you prefer to have open from the View -> Windows and Toolbars option. The default configuration has the Editor, Object inspector/Variable explorer/File explorer, and Console/History log windows open as shown above.

The Console is where python is waiting for you to type commands, which tell it to load data, do math, plot data, etc. After every command, which looks like >>> *command*, you need to hit the enter key (return key), and then python may or may not give some output. The Editor allows you to write sequences of commands, which together make up a program. The History Log stores the last 100 commands you've typed into the Console. The Object inspector/Variable explorer/File explorer windows are purely informational -- if you watch what the first two display as we go through the tutorial, you'll see that they can be quite helpful.

1.2. Variables & Types

In almost every single Python program you write, you will have variables. Variables act as placeholders for data. They can aid in short hand, as well as with logic, as variables can change, hence their name.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

Variables help programs become much more dynamic, and allow a program to always reference a value in one spot, rather than the programmer needing to repeatedly type it out, and, worse, change it if they decide to use a different definition for it.

Variables can be called just about whatever you want. You wouldn't want them to `conflict`with function names, and they also cannot start with a number.

Example 1.2:

```
weight=55
height=166
BMI=weight/(height*height)
print(BMI)
```

In this case, we will have a 0.001995935549426622printed out to console. Here, we were able to store integers and their manipulations to different variables.

We can also find out the type of any variable by using type function.

```
print(type(BMI))
```

This will return the type of variable as `<class 'float'>`

1.3. Data Structures in Python

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

1.3.1. Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

Example 1.3:

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

List indices start at 0, and lists can be sliced, concatenated and so on.

List of lists can also be created

Example 1.4:

```
weight=[55,44,45,53]
height=[166, 150, 144,155]
demo_list=[weight, height]
print(demo_list)
```

1.3.2. Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

Example 1.5:

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];

print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

When the above code is executed, it produces the following result –

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

1.3.3. Updating Lists

To update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and one can add to elements in a list with the append() method. For example –

Example 1.6:

```
list = ['physics', 'chemistry', 1997, 2000];

print ("Value available at index 2 : ")
print (list[2])
list[2] = 2001;
print ("New value available at index 2:")
print(list[2])
```

When the above code is executed, it produces the following result –

```
Value available at index 2:
1997
New value available at index 2:
2001
```

1.3.4. Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hello User!'] * 4	[' Hello User!', ' Hello User!',Hello User!', ' Hello User!']	Repetition
3 in [1, 2, 3]	True	Membership

Table1.1: Basic list operations

1.3.5. Indexing, Slicing, & Matrices

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

```
L =['spam','Spam','SPAM!']
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right

L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Table 1.2: Indexing & Slicing

1.4. Functions & Packages

1.4.1. Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions*.

Let's first discuss some built-in functions

Example 1.7:

```
list = [2017, 2001, 1997, 2000]; #creates list
print(max(list)) #finds max element in list
print(round(1.77,1))  #rounds  off  the  given  argument  upto
specified digits
help(round) # returns detailed help text
```

Output of the above code is

```
2017
1.8
Help on built-in function round in module builtins:
round(...)
round(number[, ndigits]) -> number
       Round  a  number  to  a  given  precision  in  decimal  digits
(default 0 digits). This returns an int when called with one
argument, otherwise the same type as the number. ndigits may be
negative.
```

User-defined functions

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

SYNTAX FOR DEFINING A FUNCTION

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Example 1.8:

```
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return;

#Calling function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

The result of the above code is-

```
I'm first call to user defined function!
 Again second call to the same function
```

1.4.2. Methods

Methods are same as functions but they work on some objects. Everything in python is basically an object. Depending on the type of the object there are different methods.

Example 1.9:

```
family=["mom","dad","sister","brother",44,42,22,23]
print(family.index("mom"))
print(family.count(44))
print("sister".capitalize())
```

The result of the above code is-

```
0
1
Sister
```

1.4.3. Packages

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and sub-packages and sub-sub-packages, and so on.

There are different packages present for different utilities some of which are: Numpy(efficiently works with arrays), Matplotlib(used for visualizations), Scikit_learn(used for machine learning)

Import Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax:

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.

From.... Import Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax –

```
from modname import name1[, name2[, ... nameN]]
```

This statement does not import the entire module into the current namespace; it just introduces the specified item from the module fib into the global symbol table of the importing module.

1.5. Numpy Package

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

1.5.1. Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

Example 1.10:

```
import numpy as np
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Numpy also provides many functions to create arrays:

Example 1.11:

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

1.5.2. Array Indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

Example 1.12:

```
import numpy as np

# Create the following rank 3 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2
rows
# and columns 1 and 2; b is the following array of shape (2,2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
```

```
# A slice of an array is a view into the same data, so modifying
it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower
rank,
# while using only slices yields an array of the same rank as
the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an
array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

Integer Array Indexing

When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array.

Example 1.13:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to
this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
```

```
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

Boolean Array Indexing

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

Example 1.14:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger
than 2;
                     # this returns a numpy array of Booleans of
the same
                     # shape as a, where each slot of bool_idx
tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True
values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

1.5.3. Basic Statistics with Numpy

Data analysis is all about getting to know about your data. Let's take city wise survey where we have 5000 adults and ask their height and weight. So basically we will be having 2D numpy array with 5000 rows and 2 columns (height and weight), finally we can generate summarizing statistics about the data by using different numpy methods. Some of the commonly use statistical numpy methods are:

- a.sum() Array-wise sum
- a.min() Array-wise minimum value

- b.max(axis=0) Maximum value of an array row
- b.cumsum(axis=1) Cumulative sum of the elements
- a.mean() Mean
- b.median() Median
- a.corrcoef() Correlation coefficient
- np.std(b) Standard deviation

Exercise

Variables & Types

Question 1:

- Create a variable savings with the value 100.
- Check out this variable by typing print(savings) in the script.

Question 2:

- Create a variable factor, equal to 1.10.
- Use savings and factor to calculate the amount of money you end up with after 7 years. Store the result in a new variable, result.
- Print out the value of result.

Question 3:

- Calculate the product of `savings` and `factor`. Store the result in `year1`.
- What do you think the resulting type will be? Find out by printing out the type of `year1`.
- Calculate the sum of `desc(string)` and `desc` and store the result in a new variable `doubledesc`.
- Print out `doubledesc`. Did you expect this?
- savings = 100, factor = 1.1, desc = "compound interest"

Python Lists

Question 4:

- Create a list, `areas`, that contains the area of the hallway (`hall`), kitchen (`kit`), living room (`liv`), bedroom (`bed`) and bathroom (`bath`), in this order. Use the predefined variables.
- Print `areas` with the `print()` function.

Question 5:

- Finish the line of code that creates the `areas` list such that the list first contains the name of each room as a string, and then its area. More specifically, add the strings `"hallway"`, `"kitchen"` and `"bedroom"` at the appropriate locations.
- Print `areas` again; is the printout more informative this time?

```
# area variables (in square meters)
hall = 11.25
kit = 18.0
liv = 20.0
bed = 10.75
bath = 9.50
# Adapt list areas
areas = [hall, kit, "living room", liv, bed, "bathroom", bath]
# Print areas
```

Question 6:

As a data scientist, you'll often be dealing with a lot of data, and it will make sense to group some of this data.

Instead of creating a flat list containing strings and floats, representing the names and areas of the rooms in your house, you can create a list of lists. The script on the right can already give you an idea.

- Finish the list of lists so that it also contains the bedroom and bathroom data. Make sure you enter these in order!
- Print out `house`; does this way of structuring your data make more sense?
- Print out the type of `house`. Are you still dealing with a list?

```
# area variables (in square meters)
hall = 11.25
kit = 18.0
liv = 20.0
bed = 10.75
bath = 9.50

# house information as list of lists
house = [["hallway", hall],
         ["kitchen", kit],
         ["living room", liv]]

# Print out house


# Print out the type of house
```

List sub-setting

Question 7:

- Print out the second element from the `areas` list, so `11.25`.
- Subset and print out the last element of `areas`, being `9.50`. Using a negative index makes sense here!
- Select the number representing the area of the living room and print it out.

```
# Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0,
"bedroom", 10.75, "bathroom", 9.50]

# Print out second element from areas


# Print out last element from areas


# Print out the area of the living room
```

Question8:

- Using a combination of list subsetting and variable assignment, create a new variable, `eat_sleep_area`, that contains the sum of the area of the kitchen and the area of the bedroom.
- Print this new variable `eat_sleep_area`.

```
# Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0,
"bedroom", 10.75, "bathroom", 9.50]

# Sum of kitchen and bedroom area: eat_sleep_area


# Print the variable eat_sleep_area
```

Question 9:

- Use slicing to create a list, `downstairs`, that contains the first 6 elements of `areas`.
- Do a similar thing to create a new variable, `upstairs`, that contains the last 4 elements of `areas`.
- Print both `downstairs` and `upstairs` using `print()`

```
# Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0,
"bedroom", 10.75, "bathroom", 9.50]
```

```
# Use slicing to create downstairs


# Use slicing to create upstairs


# Print out downstairs and upstairs


# Alternative slicing to create downstairs1

# Alternative slicing to create upstairs1

# Print out downstairs1 and upstairs1
```

List Manipulation

Question 10:

- You did a miscalculation when determining the area of the bathroom; it's 10.50 square meters instead of 9.50. Can you make the changes?
- Make the `areas` list more trendy! Change "living room" to "chill zone".

```
# Create the areas list
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0,
"bedroom", 10.75, "bathroom", 9.50]

# Correct the bathroom area


# Change "living room" to "chill zone"
```

Question 11:

- Use the `+` operator to paste the list `["poolhouse", 24.5]` to the end of the `areas` list. Store the resulting list as `areas_1`.
- Further extend `areas_1` by adding data on your garage. Add the string `"garage"` and float `15.45`. Name the resulting list `areas_2`.

```
# Create the areas list and make some changes
areas = ["hallway", 11.25, "kitchen", 18.0, "chill zone", 20.0,
         "bedroom", 10.75, "bathroom", 10.50]

# Add poolhouse data to areas, new list is areas_1


# Add garage data to areas_1, new list is areas_2
```

Functions

Question 12:

- Use `print()` in combination with `type()` to print out the type of `var1`.
- Use `len()` to get the length of the list `var1`. Wrap it in a `print()` call to directly print it out.
- Use `int()` to convert `var2` to an integer. Store the output as `out2`.

Create variables var1 and var2

```
var1 = [1, 2, 3, 4]
var2 = True

# Print out type of var1


# Print out length of var1


# Convert var2 to an integer: out2
```

Question 13:

- Use `+` to merge the contents of `first` and `second` into a new list: `full`.
- Call `sorted()` on `full` and specify the `reverse`argument to be `True`. Save the sorted list as `full_sorted`.
- Finish off by printing out `full_sorted`.

```
# Create lists first and second
first = [11.25, 18.0, 20.0]
second = [10.75, 9.50]

# Paste together first and second: full


# Sort full in descending order: full_sorted


# Print out full_sorted
```

Methods

Question 14: String Methods

- Use the `upper()` method on `room` and store the result in `room_up`. Use the dot notation.
- Print out `room` and `room_up`. Did both change?

- Print out the number of o's on the variable `room` by calling `count()` on `room` and passing the letter `"o"` as an input to the method. We're talking about the variable `room`, not the word `"room"`!

```
# string to experiment with: room
room = "poolhouse"

# Use upper() on room: room_up


# Print out room and room_up


# Print out the number of o's in room
```

Question 15: List Methods

- Use the `index()` method to get the index of the element in `areas` that is equal to `20.0`. Print out this index.
- Call `count()` on `areas` to find out how many times `14.5` appears in the list. Again, simply print out this number.

```
# Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Print out the index of the element 20.0


# Print out how often 14.5 appears in areas
```

Question 16:

- Use `append()` twice to add the size of the poolhouse and the garage again: `24.5` and `15.45`, respectively. Make sure to add them in this order.
- Print out `areas`
- Use the `reverse()` method to reverse the order of the elements in `areas`.
- Print out `areas` once more.

```
# Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Use append twice to add poolhouse and garage size



# Print out areas
```

```
# Reverse the orders of the elements in areas


# Print out areas
```

Packages

Question 17:

- Import the `math` package. Now you can access the constant `pi` with `math.pi`.
- Calculate the circumference of the circle and store it in `C`.
- Calculate the area of the circle and store it in `A`.

```
# Definition of radius
r = 0.43

# Import the math package


# Calculate C
C =

# Calculate A
A =

# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```

Question 18:

- Perform a selective import from the `math` package where you only import the `radians` function.
- Calculate the distance travelled by the Moon over 12 degrees of its orbit. Assign the result to `dist`. You can calculate this as r∗ϕr∗ϕ, where rr is the radius and ϕϕ is the angle in radians. To convert an angle in degrees to an angle in radians, use the `radians()` function, which you just imported.
- Print out `dist`.

```
# Definition of radius
r = 192500

# Import radians function of math package
```

```
# Travel distance of Moon if 12 degrees. Store in dist.


# Print out dist
```

Numpy Package

Question 19:

- import the `numpy` package as `np`, so that you can refer to `numpy` with `np`.
- Use `np.array()` to create a Numpy array from `baseball`. Name this array `np_baseball`.
- Print out the type of `np_baseball` to check that you got it right.

```
# Create list baseball
baseball = [180, 215, 210, 210, 188, 176, 209, 200]

# Import the numpy package as np


# Create a Numpy array from baseball: np_baseball


# Print out type of np_baseball
```

Question 20:

- Create a Numpy array from `height`. Name this new array `np_height`.
- Print `np_height`.
- Multiply `np_height` with `0.0254` to convert all height measurements from inches to meters. Store the new values in a new array, `np_height_m`.
- Print out `np_height_m` and check if the output makes sense.

```
# Create height as a regular list

# Import numpy
import numpy as np

# Create a Numpy array from height: np_height


# Print out np_height


# Convert np_height to m: np_height_m
```

```
# Print np_height_m
```

Question 21:

- Create a Numpy array from the `weight` list with the correct units. Multiply by `0.453592` to go from pounds to kilograms. Store the resulting Numpy array as `np_weight_kg`.
- Use `np_height_m` and `np_weight_kg` to calculate the BMI of each player. Use the following equation:

$$BMI = \frac{weight(kg)}{height(m)^2}$$

- Save the resulting numpy array as `bmi`.
- Print out `bmi`.

```
# Createheight and weight as a regular lists

# Import numpy
import numpy as np

# Create array from height with correct units: np_height_m
np_height_m = np.array(height) * 0.0254

# Create array from weight with correct units: np_weight_kg


# Calculate the BMI: bmi


# Print out bmi
```

Question 22:

- Create a boolean Numpy array: the element of the array should be `True` if the corresponding baseball player's BMI is below 21. You can use the `<` operator for this. Name the array `light`.
- Print the array `light`.
- Print out a Numpy array with the BMIs of all baseball players whose BMI is below 21. Use `light` inside square brackets to do a selection on the `bmi` array.

```
# Create height and weight as a regular lists

# Import numpy
import numpy as np

# Calculate the BMI: bmi
```

```
np_height_m = np.array(height) * 0.0254
np_weight_kg = np.array(weight) * 0.453592
bmi = np_weight_kg / np_height_m ** 2

# Create the light array


#                 Print                 out                 light
```

2D Numpy Arrays

Question 23: First 2D numpy array

- Use `np.array()` to create a 2D Numpy array from `baseball`. Name it `np_baseball`.
- Print out the type of `np_baseball`.
- Print out the `shape` attribute of `np_baseball`. Use `np_baseball.shape`.

```
# Create baseball, a list of lists
baseball = [[180, 78.4],
            [215, 102.7],
            [210, 98.5],
            [188, 75.2]]

# Import numpy
import numpy as np

# Create a 2D Numpy array from baseball: np_baseball


# Print out the type of np_baseball
```

Question 24: Baseball data in 2D form

- Use `np.array()` to create a 2D Numpy array from `baseball`. Name it `np_baseball`.
- Print out the `shape` attribute of `np_baseball`.

```
# Createbaseball as a regular list of lists

# Import numpy package
import numpy as np

# Create a 2D Numpy array from baseball: np_baseball


# Print out the shape of np_baseball
```

Question 25: 2D Airthmetic

- You managed to get hold on the changes in weight, height and age of all baseball players. It is available as a 2D Numpy array, `update`. Add `np_baseball` and `update` and print out the result.

```
# Createbaseball as a regular list of lists

# Createupdate as 2D Numpy array

# Import numpy package
import numpy as np

# Create np_baseball (3 cols)
np_baseball = np.array(baseball)

# Print out addition of np_baseball and update
```

Basic Statistics with Numpy

Question 26:

- Create Numpy array `np_height`, that is equal to first column of `np_baseball`.
- Print out the mean of `np_height`.
- Print out the median of `np_height`

```
# Use np_baseball from previous question

# Import numpy
import numpy as np

# Create np_height from np_baseball


# Print out the mean of np_height


# Print out the median of np_height
```

Question 27:

- The code to print out the mean height is already performed in previous question. Complete the code for the median height. Replace `None` with the correct code.
- Use `np.std()` on the first column of `np_baseball` to calculate `stddev`. Replace `None` with the correct code.

- Do big players tend to be heavier? Use `np.corrcoef()` to store the correlation between the first and second column of `np_baseball` in `corr`. Replace `None` with the correct

```
# Use np_baseball from previous question 25

# Import numpy
import numpy as np

# Print mean height (first column)
avg = np.mean(np_baseball[:,0])
print("Average: " + str(avg))

# Print median height. Replace 'None'
med = None
print("Median: " + str(med))

# Print out the standard deviation on height. Replace 'None'
stddev = None
print("Standard Deviation: " + str(stddev))

# Print out correlation between first and second column. Replace
'None'
corr = None
print("Correlation: " + str(corr))
```