Lab Assignment 01

The objective of this lab assignment is to review basic concepts of the Python programming language (functions, strings, lists, dictionaries, control flow, list comprehensions) and to introduce the main data structures, functions, and methods of the `pandas` package for data analysis.

Instructions:

Complete each task by filling in the blanks (`...`) with one or more lines of code. Each task is worth **0.5 points** (out of **10 points**).

Submission:

This assignment is due **Sunday, September 06, at 11:59PM (Central Time)**.

This assignment must be submitted on Gradescope as a **PDF file** containing the completed code for each task and the corresponding output. To save your Jupyter notebook as a PDF file, go to `File > Export Notebook As > HTML` or `File > Download As > HTML`, open the HTML file and print it as a PDF file. Additionally, this assignment has a single question on Gradescope and **all pages of the PDF file** must be assigned to this question. A **0.5-point (5%) penalty** will be applied to submissions that do not follow these guidelines. For more instructions on how to submit assignments on Gradescope, see this guide.

Late submissions will be accepted within **0-12** hours after the deadline with a **0.5-point (5%) penalty** and within **12-24** hours after the deadline with a **2-point (20%) penalty**. No late submissions will be accepted more than 24 hours after the deadline.

This assignment is individual. Offering or receiving any kind of unauthorized or unacknowledged assistance is a violation of the University's academic integrity policies, will result in a grade of zero for the assignment, and will be subject to disciplinary action.

References:

- The Python Tutorial (Link)
- 10 minutes to `pandas` (Link)
- Joel Grus. *Data Science from Scratch* (2019).

Part 1: Functions

Functions in Python are defined using the keyword `def`, followed by the function name and the parenthesized list of **parameters** or **arguments**.

The statements that form the body of the function start in the next line and must be indented. The first statement of the function body can optionally be a string containing the function's documentation string or **docstring**. The use of docstrings is strongly recommended.

Most functions end with a `return` statement that returns a value from the function. Functions without a `return` statement return `None`.

In [1]:
```
def add1(x):
    """This function adds 1 to x and returns the result."""
    return x + 1
add1(2) # returns 3
```
Out[1]: 3

In [2]:
```
help(add1) # returns information about function add1
```
```
Help on function add1 in module __main__:

add1(x)
    This function adds 1 to x and returns the result.
```

Task 01 (of 20): Write a function that returns the square of `x`.

In [3]:
```
def squared(x):
    return pow(x, 2)
```
In [4]:
```
squared(3)
```
Out[4]: 9

Short **anonymous functions** can also be defined using the keyword `lambda`. **Lambda functions** can be used wherever a function can be used.

In [ ]:
```
add2 = lambda x: x + 2
add2(3) # returns 5
```

Task 02 (of 20): Write a lambda function that returns the cube of `x`.

In [5]:
```
cubed = lambda x: pow(x, 3)
```
In [6]:
```
cubed(3)
```
Out[6]: 27

Part 2: Strings

Strings in Python can be enclosed in single quotes or double quotes. The backslash symbol (`\`) can be used to escape quotes.

The `print()` function can be used to output a string and the `len()` function can be used to return the **length** of a string.

In [9]:
```
string1 = 'Hello'
print(string1)
string2 = "world!"
print(string2)
string3 = '\"Hello world!\"'
print(string3)
```
```
Hello
world!
"Hello world!"
```
In [10]:
```
len(string1) # returns 5
```
Out[10]: 5

Strings can span multiple lines using three single quotes or double quotes.

In [12]:
```
string_multi = '''Hello
world!'''
print(string_multi)
```
```
Hello
world!
```

Strings can be concatenated using the `+` operator and repeated using the `*` operator.

Task 03 (of 20): Concatenate strings `x` and `y` and repeat string `y` two times.

In [13]:
```
x = "good"
y = "bye"
xy = x + y
yy = y + y
```
In [14]:
```
print(xy)
print(yy)
```
```
goodbye
byebye
```

Strings can be **indexed**. The first character has index 0. Negative indices start counting from the right.

Strings can also be **sliced** to obtain **substrings**. For example, `x[i:j]` returns the substring of `x` that starts in position `i` and ends in, **but does not include**, position `j`. If index `i` is omitted, it defaults to 0, and if index `j` is omitted, it defaults to the size of the string.

Task 04 (of 20): Return the first character, the next-to-last character, the first three characters, and the last seven characters of string `word`.

In [18]:
```
word = "Introduction to Data Science"
first = word[0]
next_to_last = word[len(word)-1-1]
first_three = word[:3]
last_seven = word[len(word)-7:]
```
In [19]:
```
print(first)
print(next_to_last)
print(first_three)
print(last_seven)
```
```
I
c
Int
Science
```

Python strings are **immutable**; that is, they cannot be changed. Trying to assign a value to a position in a string results in an error.

In [20]:
```
word[0] = 'i' # results in an error
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-894f9d1f2d5c> in <module>
----> 1 word[0] = 'i' # results in an error

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

Part 3: Lists

Lists are one the most useful data structures in Python. Lists can be written as a comma-separated list of **items** between brackets.

The `print()` function can be used to output a list, the `len()` function can be used to return the **number of items** in a list, and the `in` operator can be used to check whether an item is in a list.

In [1]:
```
even_list = [0, 2, 4, 6, 8, 10]
print(even_list)
```
```
[0, 2, 4, 6, 8, 10]
```
In [2]:
```
len(even_list) # returns 6
```
Out[2]: 6

In [3]:
```
1 in even_list # returns False
```
Out[3]: False

Like strings, lists can be **indexed** and **sliced**.

Task 05 (of 20): Return the second item, the last item, the middle two items, and the items in even positions of list `even_list`. *Hint:* A slice can take a third parameter that specifies its **stride**.

In [11]:
```
second = even_list[1]
last = even_list[len(even_list)-1]
middle_two = even_list[2:4]
even_positions = even_list[1::2]
```
In [12]:
```
print(second)
print(last)
print(middle_two)
print(even_positions)
```
```
2
10
[4, 6]
[2, 6, 10]
```

Unlike strings, lists are **mutable**; that is, their content can be changed. It is also possible to add a new item at the end of a list using the `append()` method.

In [13]:
```
even_list.append(12) # appends 12 to end of list
even_list.append(15) # appends 15 to end of list
print(even_list)
even_list[-1] = 14 # changes last element of list
print(even_list)
even_list[-2:] = [] # removes last two elements of list
print(even_list)
```
```
[0, 2, 4, 6, 8, 10, 12, 15]
[0, 2, 4, 6, 8, 10, 12, 14]
[0, 2, 4, 6, 8, 10]
```

Lists can be **sorted** using the `sort` method (in-place) or the `sorted()` function (not-in-place)

In [14]:
```
some_list = [2, -5, 11, 8, -3]
some_list_sorted = sorted(some_list) # sort items from smallest to largest
print(some_list_sorted)
```
```
[-5, -3, 2, 8, 11]
```

Task 06 (of 20): Sort the items of list `some_list` by absolute value from largest to smallest. *Hint:* Check the parameters of the `sorted()` function.

In [17]:
```
some_list_sorted_again = sorted([abs(ele) for ele in some_list], reverse=True)
```
In [18]:
```
print(some_list_sorted_again)
```
```
[11, 8, 5, 3, 2]
```

Part 4: Dictionaries

Another useful data structure in Python are **dictionaries**, which are sets of **keys** associated with **values**. Keys must be unique and can be of any immutable type, such as strings and numbers. Dictionaries can be written as a comma-separated list of `key: value` pairs between braces.

The `print()` function can be used to output a dictionary, the `len()` function can be used to return the **number of key-value pairs** in a dictionary, the `list()` function can be used to return a list of all keys in a dictionary, and the `in` operator can be used to check whether a key is in a dictionary.

In [24]:
```
grades = {'Joe': 85, 'Ana': 97, 'Rob': 78}
print(grades)
```
```
{'Joe': 85, 'Ana': 97, 'Rob': 78}
```
In [25]:
```
len(grades) # returns 3
```
Out[25]: 3

In [26]:
```
list(grades) # Returns 'Joe', 'Ana', and 'Rob'
```
Out[26]: ['Joe', 'Ana', 'Rob']

In [27]:
```
'Sue' in grades # returns False
```
Out[27]: False

Trying to access a key that is not in a dictionary results in an error.

In [28]:
```
print(grades['Sue']) # results in an error
```
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-28-7468b91773fd> in <module>
----> 1 print(grades['Sue']) # results in an error

KeyError: 'Sue'
```

Task 07 (of 20): Change Rob's grade to 88 and add Sue to dictionary `grades`. Sue's grade is 90.

In [32]:
```
grades['Rob'] = 88
grades['Sue'] = 90
```
In [33]:
```
print(grades)
```
```
{'Joe': 85, 'Ana': 97, 'Rob': 88, 'Sue': 90}
```

Task 08 (of 20): Delete Joe from dictionary `grades` using the `del` statement.

In [34]:
```
del grades['Joe']
```
In [35]:
```
print(grades)
```
```
{'Ana': 97, 'Rob': 88, 'Sue': 90}
```

Part 5: Control Flow

As in other programming languages, we can write `if`, `while`, and `for` statements in Python.

An `if` statement can be written using the keywords `if`, `elif` (short for `else if`), and `else`.

In [36]:
```
x = 1
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```
```
Positive
```

Task 09 (of 20): Write a function, using an `if` statement, that returns `True` if `x` is even and `False` if `x` is odd.

In [41]:
```
def is_even(x):
    if x % 2 == 0:
        return True
    else:
        return False
```
In [42]:
```
print(is_even(2))
print(is_even(5))
```
```
True
False
```

A `while` statement executes as long as a condition is `True`.

In [43]:
```
x = 1
while x < 5:
    print(x)
    x = x + 1
```
```
1
2
3
4
```

Task 10 (of 20): Write a `while` statement that prints and then squares `x` as long as `x` is less than 100.

In [44]:
```
x = 2
while x < 100:
    print(x)
    x = pow(x,2)
```
```
2
4
16
```

A `for` statement iterates over the items of a sequence, such as a list or a string, in the order that they appear in the sequence.

In [45]:
```
words = ['introduction', 'to', 'data', 'science']
for w in words:
    print(w, len(w))
```
```
introduction 12
to 2
data 4
science 7
```

Task 11 (of 20): Write a `for` statement that iterates over the characters in string `long_word` and prints those that are vowels.

In [47]:
```
long_word = "computation"
for w in long_word:
    if w == 'v' or w == 'o' or w == 'w' or w == 'e' or w == 'l' or w == 's':
        print(w)
```
```
o
o
```

A `for` statement can also be used to iterate over the key-value pairs in a dictionary.

In [48]:
```
for student, grade in grades.items():
    print("The grade of", student, "is", grade)
```
```
The grade of Ana is 97
The grade of Rob is 88
The grade of Sue is 90
```

To iterate over a sequence of numbers, the `range()` function can be used. For example, `range(10)` returns a sequence from 0 to 9 and `range(5, 10)` returns a sequence from 5 to 9.

Task 12 (of 20): Write a `for` statement, using the `range()` function, that iterates over the first 10 positive integers and prints those that are multiples of 3.

In [53]:
```
for i in range(0,10):
    if i % 3 == 0 and i != 0:
        print(i)
```
```
3
6
9
```

Part 6: List Comprehensions

List comprehensions provide a concise way to create a list where each item satisfies a certain condition and/or is the result of an operation applied to the items of another list.

A list comprehension is written between brackets and contains an expression and one or more `for` statements followed by zero or more `if` statements.

In [54]:
```
odd_list = [x for x in range(10) if x % 2 != 0]
print(odd_list)
```
```
[1, 3, 5, 7, 9]
```

Task 13 (of 20): Write a list comprehension that creates a list containing the squares of the items in list `odd_list`.

In [55]:
```
odd_squared_list = [pow(x,2) for x in odd_list]
print(odd_squared_list)
```
```
[1, 9, 25, 49, 81]
```

Task 14 (of 20): Write a list comprehension that creates a list containing all pairs of integers `(x, y)` where $0 \leq$ `x` ≤ 3 and `x` $\leq$ `y` ≤ 3. For example, `(0, 0)` and `(1, 3)` should be in the list, but `(3, 1)` should not. *Hint:* Use two `for` statements and the `range()` function.

In [71]:
```
pairs_list = [(x, y) for x in range(0,4) for y in range(x,4)]
print(pairs_list)
```
```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Part 7: pandas - Data Structures

`pandas` is a Python package for **data analysis**. It is well suited for analyzing tabular data, such as SQL tables or Excel spreadsheets, and it provides functions and methods for easily manipulating (reshaping, slicing, merging, etc.) datasets.

pandas has two primary data structures: `Series` and `DataFrames`. A `Series` is a one-dimensional homogeneously-typed array and a `DataFrame` is a two-dimensional potentially heterogeneously-typed table.

In [72]:
```
import numpy as np
import pandas as pd
```

A `Series` can be created by passing a list of values.

In [73]:
```
s = pd.Series([0, 2, 4, 8, 10])
s
```
Out[73]:
```
0     0
1     2
2     4
3     8
4    10
dtype: int64
```

A `DataFrame` can be created by passing a dictionary.

In [74]:
```
df = pd.DataFrame({'name': ['Joe', 'Ana', 'Rob', 'Sue'],
                   'age': [24, 21, 25, 24],
                   'grade': [85.0, 97.0, 78.0, 90.0],
                   'major': ['Math', 'CS', 'CS', 'ECE']})
df
```
Out[74]:

	name	age	grade	major
0	Joe	24	85.0	Math
1	Ana	21	97.0	CS
2	Rob	25	78.0	CS
3	Sue	24	90.0	ECE

The columns of a `DataFrame` can have different types and can be displayed using the `columns` method.

In [75]:
```
df.dtypes
```
Out[75]:
```
name      object
age        int64
grade    float64
major     object
dtype: object
```
In [76]:
```
df.columns
```
Out[76]: Index(['name', 'age', 'grade', 'major'], dtype='object')

Selecting a single column of a `DataFrame` yields a `Series`.

In [77]:
```
df['name']
```
Out[77]:
```
0    Joe
1    Ana
2    Rob
3    Sue
Name: name, dtype: object
```

A subset of rows and columns can also be selected using the `iloc` and `loc` methods.

Task 15 (of 20): Select the first two rows and the last two columns of `DataFrame` `df` using the `iloc` method. *Hint:* The `iloc` method is used for indexing by integer position.

In [92]:
```
df.iloc[[0,1], [2,3]]
# df.iloc[:2, 2:]
```
Out[92]:

	grade	major
0	85.0	Math
1	97.0	CS

Task 16 (of 20): Select the first two rows and the last two columns of `DataFrame` `df` using the `loc` method. *Hint:* The `loc` method is used for indexing by label.

In [107]:
```
df.loc[:1,'grade':'major']
```
Out[107]:

	grade	major
0	85.0	Math
1	97.0	CS

Part 8: pandas - Sorting, Grouping, and Merging

The values in a `DataFrame` can be **sorted** using the `sort_values` method.

Task 17 (of 20): Sort the rows of `DataFrame` `df` by grade from largest to smallest using the `sort_values` method. *Hint:* Check the parameters of the `sort_values` method.

In [110]:
```
df.sort_values(by='grade', ascending=False)
```
Out[110]:

	name	age	grade	major
1	Ana	21	97.0	CS
3	Sue	24	90.0	ECE
0	Joe	24	85.0	Math
2	Rob	25	78.0	CS

The values in a `DataFrame` can also be **grouped** based on some criteria using the `groupby` method. Then, a function can be applied to each group independently.

Task 18 (of 20): Group the rows of `DataFrame` `df` by major using the `groupby` method and find the mean age and mean grade of each group.

In [113]:
```
df.groupby(by='major').mean()
```
Out[113]:

	age	grade
major		
CS	23	87.5
ECE	24	90.0
Math	24	85.0

`DataFrames` can be **concatenated** together using the `concat()` function.

In [114]:
```
df2 = pd.DataFrame({'name': ['Tom'],
                    'age': [22],
                    'grade': [88.0],
                    'major': ['Math']})
df2
```
Out[114]:

	name	age	grade	major
0	Tom	22	88.0	Math

Task 19 (of 20): Concatenate `DataFrames` `df` and `df2` using the `concat()` function.

In [117]:
```
pd.concat([df,df2])
```
Out[117]:

	name	age	grade	major
0	Joe	24	85.0	Math
1	Ana	21	97.0	CS
2	Rob	25	78.0	CS
3	Sue	24	90.0	ECE
0	Tom	22	88.0	Math

Alternatively, rows can be added to a `DataFrame` using the `append` method.

Task 20 (of 20): Add `DataFrame` `df2` to `DataFrame` `df` using the `append()` method.

In [118]:
```
df.append(df2)
```
Out[118]:

	name	age	grade	major
0	Joe	24	85.0	Math
1	Ana	21	97.0	CS
2	Rob	25	78.0	CS
3	Sue	24	90.0	ECE
0	Tom	22	88.0	Math