Python Code for QSS Chapter 1: Introduction

Kosuke Imai, Python code by Jeff Allen

First Printing

Section 1.1: Overview of the Book

Section 1.2: How to Use this Book

Section 1.3: Introduction to Python and Pandas

Section 1.3.1: Arithmetic Operations: Python as a Calculator

[ ]:
```
5 + 3
```

[ ]: 8

[ ]:
```
5 - 3
```

[ ]: 2

[ ]:
```
5 / 3
```

[ ]: 1.6666666666666667

[ ]:
```
5 ** 3
```

[ ]: 125

[ ]:
```
5 * (10 - 3)
```

[ ]: 35

[ ]:
```
from math import sqrt

sqrt(4)
```

[ ]: 2.0

Section 1.3.2: Modules and Packages

[ ]:
```
# earlier, we imported the sqrt function from the math module

# we can import the whole module
import math
```

```python
# use the dot notation to access functions
math.sqrt(4)
```

[ ]: 2.0

```python
math.log(1)
```

[ ]: 0.0

External packages are typically installed from public repositories, such as the Python Package Index (PyPI) and conda-forge, as follows:

- From PyPI: `pip install <package_name>`
- From conda-forge: `conda install <package_name>`

A good practice is to install required packages into a virtual environment.

```python
# import the external package Pandas and give it the conventional alias `pd`
import pandas as pd
```

Section 1.3.3: Python Scripts

```python
'''
This is the start of a Python script
The heading provides some information about the file

File name: testing_arithmetic.py
Author: <Your Name>
Date created: <Date>
Purpose: Practicing basic math commands and commenting in Python
'''

import math

5-3 # What is 5 minus 3?
5/3
5**3
5 * (10 - 3) # A bit more complex

# This function returns the square root of a number
math.sqrt(4)
```

[ ]: 2.0

Section 1.3.4: Variables and Objects

[ ]:
```python
# assign the sum of 5 and 3 to the variable result
result = 5 + 3
result
```

[ ]: 8

[ ]:
```python
print(result)
```

8

[ ]:
```python
result = 5 - 3
result
```

[ ]: 2

[ ]:
```python
kosuke = 'instructor'
kosuke
```

[ ]: 'instructor'

[ ]:
```python
kosuke = 'instructor and author'
kosuke
```

[ ]: 'instructor and author'

[ ]:
```python
Result = '5'
Result
```

[ ]: '5'

[ ]:
```python
result
```

[ ]: 2

[ ]:
```python
# add 5 to the variable result
result+=5
result
```

[ ]: 7

Python is an object-oriented programming (OOP) language. Everything in Python is an object of a certain class. Section 3.7.2 discusses classes and objects in more detail. While we do not cover OOP techniques in this book, we interact with classes and objects throughout.

[ ]:
```python
type(result)
```

[ ]: int

```
[ ]: type(Result)
```

```
[ ]: str
```

```
[ ]: type(math.sqrt)
```

```
[ ]: builtin_function_or_method
```

Section 1.3.4: Python Data Structures: Lists, Dictionaries, Tuples, Sets

Lists, dictionaries, tuples, and sets are built-in Python data structures. In this book, we primarily use them to manipulate inputs to and outputs from data structures and models in external packages. However, understanding them is critical for effective Python use. External packages come and go. Knowing how to use the built-in data structures enables you to pick up new data analysis packages and infrastructure paradigms more easily.

List

```
[ ]: world_pop = [2525779, 3026003, 3691173, 4449049, 5320817, 6127700, 6916183]
     world_pop
```

```
[ ]: [2525779, 3026003, 3691173, 4449049, 5320817, 6127700, 6916183]
```

```
[ ]: pop_first = [2525779, 3026003, 3691173]
     pop_second = [4449049, 5320817, 6127700, 6916183]
     pop_all = pop_first + pop_second
     pop_all
```

```
[ ]: [2525779, 3026003, 3691173, 4449049, 5320817, 6127700, 6916183]
```

```
[ ]: # Python uses zero-based indexing
     world_pop[0] # the first observation
```

```
[ ]: 2525779
```

```
[ ]: world_pop[1] # the second observation
```

```
[ ]: 3026003
```

```
[ ]: world_pop[-1] # the last observation
```

```
[ ]: 6916183
```

```
[ ]: # Python uses "up to but not including" slicing semantics
     world_pop[0:4] # the first four observations
```

```
[ ]: [2525779, 3026003, 3691173, 4449049]
```

```
[ ]: world_pop[:4] # an alternative was to slice the first four observations
```

[ ]: [2525779, 3026003, 3691173, 4449049]

[ ]:
```python
# How many observations are in the list?
len(world_pop)
```

[ ]: 7

[ ]:
```python
# Select the last three observations
world_pop[4:]
```

[ ]: [5320817, 6127700, 6916183]

[ ]:
```python
# return a sequence of decades as a list using range(start, stop, step)
list(range(1950, 2011, 10)) # specify 2011 to include 2010
```

[ ]: [1950, 1960, 1970, 1980, 1990, 2000, 2010]

[ ]:
```python
# A list can contain different data types
mixed_list = [10, 10.5, True, 'USA', 'Canada']
mixed_list
```

[ ]: [10, 10.5, True, 'USA', 'Canada']

[ ]:
```python
# Lists are mutable
mixed_list[4] = 'Mexico'
mixed_list
```

[ ]: [10, 10.5, True, 'USA', 'Mexico']

[ ]:
```python
mixed_list.append('Canada')
mixed_list
```

[ ]: [10, 10.5, True, 'USA', 'Mexico', 'Canada']

[ ]:
```python
# Assignment with mutable objects can be tricky because variables are pointers
alt_list = mixed_list
alt_list
```

[ ]: [10, 10.5, True, 'USA', 'Mexico', 'Canada']

[ ]:
```python
mixed_list.append('USA')
mixed_list
```

[ ]: [10, 10.5, True, 'USA', 'Mexico', 'Canada', 'USA']

[ ]:
```python
alt_list # alt_list is also updated!
```

[ ]: [10, 10.5, True, 'USA', 'Mexico', 'Canada', 'USA']

```
[ ]: # the .copy() method overrides this behavior
alt_list = mixed_list.copy()
alt_list
```

```
[ ]: [10, 10.5, True, 'USA', 'Mexico', 'Canada', 'USA']
```

```
[ ]: mixed_list.append(10)
mixed_list
```

```
[ ]: [10, 10.5, True, 'USA', 'Mexico', 'Canada', 'USA', 10]
```

```
[ ]: alt_list # alt_list is not updated
```

```
[ ]: [10, 10.5, True, 'USA', 'Mexico', 'Canada', 'USA']
```

Built-in Python data structures are generally not vectorized. For example, multiplying world_pop by 2 will concatenate the list twice, rather than multiply each element by 2.

```
[ ]: world_pop*2
```

```
[ ]: [2525779,
 3026003,
 3691173,
 4449049,
 5320817,
 6127700,
 6916183,
 2525779,
 3026003,
 3691173,
 4449049,
 5320817,
 6127700,
 6916183]
```

```
[ ]: # We can use a 'list comprehension' to perform element-wise arithmetic
pop_million = [pop / 1000 for pop in world_pop]
pop_million
```

```
[ ]: [2525.779, 3026.003, 3691.173, 4449.049, 5320.817, 6127.7, 6916.183]
```

In this book, we use pandas and numpy to conduct vectorized operations. Nevertheless, list comprehensions are very useful for returning observations that meet certain conditions.

```
[ ]: # return strings from mixed_list
[item for item in mixed_list if isinstance(item, str)]
```

```
[ ]: ['USA', 'Mexico', 'Canada', 'USA']
```

Dictionary

[ ]:
```python
# dictionaries are key-value pairs; they need not be ordered
world_pop_dict = {'1950': 2525779, '1960': 3026003, '1970': 3691173,
                  '1980': 4449049, '1990': 5320817, '2000': 6127700}
world_pop_dict
```

[ ]:
```
{'1950': 2525779,
 '1960': 3026003,
 '1970': 3691173,
 '1980': 4449049,
 '1990': 5320817,
 '2000': 6127700}
```

[ ]:
```python
world_pop_dict['1990']
```

[ ]:
```
5320817
```

[ ]:
```python
# add a new key-value pair
world_pop_dict['2010'] = 6916183
world_pop_dict
```

[ ]:
```
{'1950': 2525779,
 '1960': 3026003,
 '1970': 3691173,
 '1980': 4449049,
 '1990': 5320817,
 '2000': 6127700,
 '2010': 6916183}
```

[ ]:
```python
# return the keys
world_pop_dict.keys()
```

[ ]:
```
dict_keys(['1950', '1960', '1970', '1980', '1990', '2000', '2010'])
```

[ ]:
```python
# return the values
world_pop_dict.values()
```

[ ]:
```
dict_values([2525779, 3026003, 3691173, 4449049, 5320817, 6127700, 6916183])
```

[ ]:
```python
# return the values as a list
list(world_pop_dict.values())
```

[ ]:
```
[2525779, 3026003, 3691173, 4449049, 5320817, 6127700, 6916183]
```

Tuple

[ ]:
```python
# tuples are like lists, but they are immutable
world_pop_t = (2525779, 3026003, 3691173, 4449049, 5320817, 6127700, 6916183)
world_pop_t
```

[ ]: (2525779, 3026003, 3691173, 4449049, 5320817, 6127700, 6916183)

[ ]:
```
world_pop_t[0]
```

[ ]: 2525779

[ ]:
```
world_pop_t[4:]
```

[ ]: (5320817, 6127700, 6916183)

[ ]:
```
# we cannot change the values of a tuple; this will raise an error:
# world_pop_t[2] = 100
```

Set

[ ]:
```
# a set contains only unique values
print(mixed_list)

set(mixed_list)
```

```
[10, 10.5, True, 'USA', 'Mexico', 'Canada', 'USA', 10]
```

[ ]: {10, 10.5, 'Canada', 'Mexico', True, 'USA'}

Section 1.3.5: Pandas Data Structures: Series and DataFrames

Series

[ ]:
```
# a series is a vector with an index
world_pop_s = pd.Series(world_pop)
world_pop_s
```

[ ]:
```
0    2525779
1    3026003
2    3691173
3    4449049
4    5320817
5    6127700
6    6916183
dtype: int64
```

[ ]:
```
# selection and slicing are similar to lists
world_pop_s[1]
world_pop_s[:4]
```

[ ]:
```
0    2525779
1    3026003
2    3691173
3    4449049
```

```
dtype: int64
```

[ ]:
```
# select non-consecutive observations
world_pop_s[[0, 2]]
```

[ ]:
```
0    2525779
2    3691173
dtype: int64
```

[ ]:
```
# select everything except the third observation
world_pop_s.drop(2)
```

[ ]:
```
0    2525779
1    3026003
3    4449049
4    5320817
5    6127700
6    6916183
dtype: int64
```

[ ]:
```
# select everything except the last observation
world_pop_s[:-1]
```

[ ]:
```
0    2525779
1    3026003
2    3691173
3    4449049
4    5320817
5    6127700
dtype: int64
```

[ ]:
```
# the index is flexible
world_pop_s2 = pd.Series(world_pop_dict)
world_pop_s2
```

[ ]:
```
1950    2525779
1960    3026003
1970    3691173
1980    4449049
1990    5320817
2000    6127700
2010    6916183
dtype: int64
```

[ ]:
```
world_pop_s2['1990']
```

[ ]:
```
5320817
```

[ ]:
```
# series are vectorized
pop_million = world_pop_s / 1000
pop_million
```

[ ]:
```
0    2525.779
1    3026.003
2    3691.173
3    4449.049
4    5320.817
5    6127.700
6    6916.183
dtype: float64
```

[ ]:
```
pop_rate = world_pop_s / world_pop_s[0]
pop_rate
```

[ ]:
```
0    1.000000
1    1.198047
2    1.461400
3    1.761456
4    2.106604
5    2.426063
6    2.738238
dtype: float64
```

Vector arithmetic with series requires that the indices match. One way to ensure this is to reset the index after slicing the series. Specify `drop=True` in `reset_index` to avoid adding the old index as a column.

[ ]:
```
pop_increase = (
    world_pop_s.drop(0).reset_index(drop=True) -
    world_pop_s.drop(6).reset_index(drop=True)
)
pop_increase
```

[ ]:
```
0    500224
1    665170
2    757876
3    871768
4    806883
5    788483
dtype: int64
```

[ ]:
```
percent_increase = pop_increase / world_pop_s.drop(6) * 100
percent_increase
```

[ ]:
```
0    19.804741
1    21.981802
```

```
2    20.532118
3    19.594480
4    15.164645
5    12.867520
dtype: float64
```

[ ]:
```
# series have many useful methods that help perform calculations
world_pop_s.pct_change() * 100
```

[ ]:
```
0          NaN
1    19.804741
2    21.981802
3    20.532118
4    19.594480
5    15.164645
6    12.867520
dtype: float64
```

[ ]:
```
# series are mutable
percent_increase[[0,1]] = [20, 22]
percent_increase
```

[ ]:
```
0    20.000000
1    22.000000
2    20.532118
3    19.594480
4    15.164645
5    12.867520
dtype: float64
```

DataFrame

[ ]:
```
world_pop_df = pd.DataFrame(
    # build the data frame from a dictionary of lists
    {'year': list(range(1950, 2011, 10)),
    'pop': world_pop}
)

world_pop_df
```

[ ]:

	year	pop
0	1950	2525779
1	1960	3026003
2	1970	3691173
3	1980	4449049
4	1990	5320817
5	2000	6127700
6	2010	6916183

[ ]:
```
world_pop_df.columns
```

[ ]:
```
Index(['year', 'pop'], dtype='object')
```

[ ]:
```
world_pop_df.shape
```

[ ]:
```
(7, 2)
```

[ ]:
```
world_pop_df.describe()
```

[ ]:

	year	pop
count	7.000000	7.000000e+00
mean	1980.000000	4.579529e+06
std	21.602469	1.625004e+06
min	1950.000000	2.525779e+06
25%	1965.000000	3.358588e+06
50%	1980.000000	4.449049e+06
75%	1995.000000	5.724258e+06
max	2010.000000	6.916183e+06

[ ]:
```
# display the summary as integers
world_pop_df.describe().astype(int)
```

[ ]:

	year	pop
count	7	7
mean	1980	4579529
std	21	1625003
min	1950	2525779
25%	1965	3358588
50%	1980	4449049
75%	1995	5724258
max	2010	6916183

[ ]:
```
# extract the 'pop' column; returns a series
world_pop_df['pop']
```

[ ]:
```
0    2525779
1    3026003
2    3691173
3    4449049
4    5320817
5    6127700
6    6916183
Name: pop, dtype: int64
```

[ ]:
```
# extract the first three rows (and all columns)
world_pop_df[:3]
```

```
[ ]:    year      pop
     0  1950  2525779
     1  1960  3026003
     2  1970  3691173
```

To select a mixture of rows and columns, use the `.loc` and `.iloc` methods. The former enables selection with labels, while the latter enables selection with integers.

```python
[ ]: # select the first three rows and the 'pop' column
     world_pop_df.loc[:2, 'pop']
```

```
[ ]: 0    2525779
     1    3026003
     2    3691173
     Name: pop, dtype: int64
```

```python
[ ]: # select the first three rows and both columns (but switch the column order)
     world_pop_df.loc[:2, ['pop', 'year']]
```

```
[ ]:        pop  year
     0  2525779  1950
     1  3026003  1960
     2  3691173  1970
```

Notice that with .loc, the last index is included. This differs from typical Python slicing semantics. The reason is that .loc is a label-based method of selection.

```python
[ ]: # select the first three rows and the 'pop' column using integers
     world_pop_df.iloc[:3, 1] # now the last index is excluded
```

```
[ ]: 0    2525779
     1    3026003
     2    3691173
     Name: pop, dtype: int64
```

```python
[ ]: # select the first three rows and both columns (but switch the column order)
     world_pop_df.iloc[:3, [1, 0]]
```

```
[ ]:        pop  year
     0  2525779  1950
     1  3026003  1960
     2  3691173  1970
```

```python
[ ]: # take elements 1, 3, 5, ... of the 'pop' variable using step size 2
     world_pop_df['pop'][::2]
```

```
[ ]: 0    2525779
     2    3691173
     4    5320817
```

```
6    6916183
Name: pop, dtype: int64
```

[ ]:
```
# concatenate 'pop' with an NA; use numpy to generate the NA
import numpy as np

world_pop = pd.concat([world_pop_df['pop'], pd.Series([np.nan])],
                      ignore_index=True)

world_pop
```

[ ]:
```
0    2525779.0
1    3026003.0
2    3691173.0
3    4449049.0
4    5320817.0
5    6127700.0
6    6916183.0
7          NaN
dtype: float64
```

[ ]:
```
# pandas ignores NA values by default
world_pop.mean().round(2)
```

[ ]:
```
4579529.14
```

[ ]:
```
# we can override this behavior
world_pop.mean(skipna=False)
```

[ ]:
```
nan
```

Section 1.3.6: Functions and Methods

[ ]:
```
world_pop = world_pop_df['pop']
world_pop
```

[ ]:
```
0    2525779
1    3026003
2    3691173
3    4449049
4    5320817
5    6127700
6    6916183
Name: pop, dtype: int64
```

[ ]:
```
len(world_pop)
```

[ ]:
```
7
```

[ ]:
```
# methods are functions that are attached to objects
world_pop.min() # access methods using the dot notation
```

[ ]: 2525779

[ ]:
```
world_pop.max()
```

[ ]: 6916183

[ ]:
```
world_pop.mean()
```

[ ]: 4579529.142857143

[ ]:
```
# round the result
world_pop.mean().round(2)
```

[ ]: 4579529.14

[ ]:
```
world_pop.sum() / len(world_pop)
```

[ ]: 4579529.142857143

[ ]:
```
# Use numpy to generate a sequence of decades
year = np.arange(1950, 2011, 10)
year
```

[ ]: array([1950, 1960, 1970, 1980, 1990, 2000, 2010])

[ ]:
```
np.arange(start=1950, step=10, stop=2011)
```

[ ]: array([1950, 1960, 1970, 1980, 1990, 2000, 2010])

[ ]:
```
# reverse sequence and convert to a series
pd.Series(np.arange(2010, 1949, -10))
```

[ ]:
```
0    2010
1    2000
2    1990
3    1980
4    1970
5    1960
6    1950
dtype: int32
```

[ ]:
```
world_pop.index
```

[ ]: RangeIndex(start=0, stop=7, step=1)

[ ]:
```
list(world_pop.index)
```

[ ]: [0, 1, 2, 3, 4, 5, 6]

```
[ ]: # set the index to the year
world_pop.index = year
world_pop.index
```

[ ]: Index([1950, 1960, 1970, 1980, 1990, 2000, 2010], dtype='int32')

```
[ ]: world_pop
```

```
[ ]: 1950    2525779
1960    3026003
1970    3691173
1980    4449049
1990    5320817
2000    6127700
2010    6916183
Name: pop, dtype: int64
```

```
[ ]: '''
def myfunction(input1, input2, ..., inputN):
    DEFINE `output` USING INPUTS
    return output
'''

def my_summary(x): # function takes one input
    s_out = x.sum()
    l_out = len(x)
    m_out = x.mean()
    # define the output
    out = pd.Series([s_out, l_out, m_out], index=['sum', 'length', 'mean'])
    return out # end function by calling output
```

```
[ ]: z = np.arange(1, 11)

my_summary(z)
```

```
[ ]: sum       55.0
length    10.0
mean       5.5
dtype: float64
```

```
[ ]: my_summary(world_pop).astype(int) # return summary as integers
```

```
[ ]: sum       32056704
length           7
mean       4579529
dtype: int32
```

[ ]:
```
type(my_summary) # functions are objects
```

[ ]:
```
function
```

[ ]:
```
type(np.arange)
```

[ ]:
```
builtin_function_or_method
```

Section 1.3.7: Loading and Saving Data

Many modern IDEs enable you to set up a workspace or a project that automatically configures the working directory and enables you to work with relative paths, rather than setting the working directory manually. In cases where you need to manipulate the working directory, use the `os` module.

[ ]:
```
import os

# get the current working directory
# os.getcwd()

# change the working directory
# os.chdir('<path_name>')
```

[ ]:
```
# Import a CSV
un_pop = pd.read_csv('UNpop.csv')
un_pop
```

[ ]:

	year	world_pop
0	1950	2525779
1	1960	3026003
2	1970	3691173
3	1980	4449049
4	1990	5320817
5	2000	6127700
6	2010	6916183

[ ]:
```
# Import a DTA
un_pop_dta = pd.read_stata('UNpop.dta')
un_pop_dta
```

[ ]:

	year	world_pop
0	1950	2525779
1	1960	3026003
2	1970	3691173
3	1980	4449049
4	1990	5320817
5	2000	6127700
6	2010	6916183

Pandas supports reading a wide variety of file types.

```
[ ]: # Write to a CSV
     un_pop.to_csv('UNpop.csv', index=False)
```

```
[ ]: # Write to a pickle; a pickle serializes the data
     un_pop.to_pickle('UNpop.pkl')
```