Practical Business Python

Lecture 3: Built-In Data Structures, Functions, and Files.

Iegor Vyshnevskyi

Woosong University

September 21, 2023

Agenda

1. Python Data Structure:
    A. Tuple
    B. List
    C. Dictionary
    D. Set
2. Python Functions
3. Files and the Operating System
4. In-class Assignment

1. Python Data Structure

Python’s data structures are simple but powerful.

Mastering their use is a critical part of becoming a proficient Pythonista.

We start with tuple, list, and dictionary, which are some of the most frequently used sequence types.

A. Python Tuple

Tuple basics

Tuple [ˈtjuːpəl] is a fixed-length, immutable sequence of Python objects which, once assigned, cannot be changed.

In some ways, a tuple is similar to a Python list in terms of indexing, nested objects, and repetition but the main difference between both is Python tuple is immutable, unlike the Python list which is mutable.

The easiest way to create one is with a comma-separated sequence of values wrapped in parentheses:

```
mytuple = ("apple", "banana", "cherry")
```

```
>>> mytuple = ("apple", "banana", "cherry")
```

Source: Wes McKinney (2022)

Tuple basics

Tuple items are ordered, unchangeable, and allow duplicate values.

- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Since tuples are indexed, they can have items with the same value.

```
# Example
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

```
>>> thistuple = ("apple", "banana", "cherry", "apple", "cherry")
>>> print(thistuple)
```

Source: W3Schools

Tuple basics

Tuple Length

- to determine how many items a tuple has, use the len() function:

```
# Example
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(len(thistuple))
```

Tuple With One Item

- To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple (use type() function to check the type):

```
# Example
thistuple = ("apple",)
print(type(thistuple))
```

Tuple Items - Data Types

- Tuple items can be of any data type:

```
# Example
# String, int and boolean data types, and mixed data types:
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
tuple4 = ("abc", 34, True, 40, "male")
```

Source: W3Schools

Tuple basics

The tuple() Constructor

- It is also possible to use the tuple() constructor to make a tuple:

```
# Example
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

Tuple Indexing

- We can use the index operator [] to access an item in a tuple, where the index starts from 0.
- The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Example
tuple4 = ("abc", 34, True, 40, "male")
print(tuple4[0])
print(tuple4[-1])
```

```
>>> tuple4 = ("abc", 34, True, 40, "male")
>>> print(tuple4[0])
abc
>>> print(tuple4[-1])
male
```

Source: W3Schools; Wes McKinney (2022); Programiz

Tuple basics

Tuple Slicing

- We can access a range of items in a tuple by using the slicing operator colon ‘:’.

```
# accessing tuple elements using slicing
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
# elements 2nd to 4th index
print(my_tuple[1:4])
# elements beginning to 2nd
print(my_tuple[:-7])
# elements 8th to end
print(my_tuple[7:])
# elements beginning to end
print(my_tuple[:])
```

```
>>> my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
>>> # elements 2nd to 4th index
>>>
>>> print(my_tuple[1:4])
('r', 'o', 'g')
>>> # elements beginning to 2nd
>>>
>>> print(my_tuple[:-7])
('p', 'r')
>>> # elements 8th to end
>>>
>>> print(my_tuple[7:])
('i', 'z')
>>> # elements beginning to end
>>>
>>> print(my_tuple[:])
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Source: Programiz

Tuple basics

Python Tuple manipulation and Methods

- In Python ,methods that add items or remove items are not available with tuple.
- You can concatenate tuples using the '+' operator to produce longer tuples:

```
# Example
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
tuple2_3 = tuple2 + tuple3
print(tuple2_3)
```

```
>>> tuple2 = (1, 5, 7, 9, 3)
>>> tuple3 = (True, False, False)
>>> tuple2_3 = tuple2 + tuple3
>>> print(tuple2_3)
(1, 5, 7, 9, 3, True, False, False)
```

- Multiplying a tuple by an integer, as with lists, has the effect of concatenating that many copies of the tuple:

```
>>> tuple2 = (1, 5, 7, 9, 3)
>>> tuple2_2 = tuple2 * 2
>>> print(tuple2_2)
(1, 5, 7, 9, 3, 1, 5, 7, 9, 3)
```

```
# Example
tuple2 = (1, 5, 7, 9, 3)
tuple2_2 = tuple2 * 2
print(tuple2_2)
```

- You can count and index tuple's elements.

```
# Example
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple.count('r'))
print(my_tuple.index('o'))
```

```
>>> my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
>>> print(my_tuple.count('r'))
2
>>> print(my_tuple.index('o'))
2
```

Source: Wes McKinney (2022); Programiz

Tuple basics

Advantages of Tuple over List in Python

Since tuples are quite similar to lists, both of them are used in similar situations.

However, there are certain advantages of implementing a tuple over a list:

- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with a list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Source: Programiz

B. Python List

List basics

In contrast with tuples, ***list***s are variable length and their contents can be modified in place. Lists are mutable.

As opposed to int, bool etc., a list is a compound data type; you can group values together. You can define them using square brackets [] or using the list type function.

```
# Example
a = "is"
b = "short"
my_list = ["this", "list", a, b]
print(my_list)
my_list = list(("this", "list", a, b)) # note the double round-brackets
print(my_list)
```

```
>>> a = "is"
>>> b = "short"
>>> my_list = ["this", "list", a, b]
>>> print(my_list)
['this', 'list', 'is', 'short']
>>> my_list = list(("this", "list", a, b)) # note the double round-brackets
>>> print(my_list)
['this', 'list', 'is', 'short']
```

Source: DataCamp; Wes McKinney (2022)

List basics

Example

- After measuring the height of your family, you decide to collect some information on the house you're living in.
- The areas of the different parts of your house are stored in separate variables for now, as shown in the script.

```
# area variables (in square meters)
hall = 12.45
kit = 17.0
liv = 30.0
bed = 13.55
bath = 11.50
# Create list areas
areas = [hall, kit, liv, bed, bath]
# Print areas
print(areas)
```

```
>>> hall = 12.45
>>> kit = 17.0
>>> liv = 30.0
>>> bed = 13.55
>>> bath = 11.50
>>> # Create list areas
>>>
>>> areas = [hall, kit, liv, bed, bath]
>>> # Print areas
>>>
>>> print(areas)
[12.45, 17.0, 30.0, 13.55, 11.5]
```

Source: DataCamp; Wes McKinney (2022)

List basics

- A list has the flexibility to accommodate various Python data types. While it's relatively uncommon, a list can even include a combination of Python types such as strings, floats, booleans, and more.

Example

```
# Example
# area variables (in square meters)
hall = 12.45
kit = 17.0
liv = 30.0
bed = 13.55
bath = 11.50
# Adapt list areas
areas = ["hallway", hall, "kitchen", kit, "living room", liv, "bedroom", bed, "bathroom", bath]
# Print areas
print(areas)
```

```
>>> hall = 12.45
>>> kit = 17.0
>>> liv = 30.0
>>> bed = 13.55
>>> bath = 11.50
>>> # Adapt list areas
>>>
>>> areas = ["hallway", hall, "kitchen", kit, "living room", liv, "bedroom", bed, "bathroom", bath]
>>> # Print areas
>>>
>>> print(areas)
['hallway', 12.45, 'kitchen', 17.0, 'living room', 30.0, 'bedroom', 13.55, 'bathroom', 11.5]
```

Source: DataCamp; Wes McKinney (2022)

List basics

List of lists

- Rather than forming a single list that combines strings and floats to represent room names and their respective areas in your house, you have the option to construct a list of lists.

Example

```
>>> hall = 12.45
>>> kit = 17.0
>>> liv = 30.0
>>> bed = 13.55
>>> bath = 11.50
>>> # house information as list of lists
>>>
>>> house = [["hallway", hall],
...          ["kitchen", kit],
...          ["living room", liv],
...          ["bedroom", bed],
...          ["bathroom", bath]]
>>> # Print out house
>>>
>>> print(house)
[['hallway', 12.45], ['kitchen', 17.0], ['living room', 30.0], ['bedroom', 13.55], ['bathroom', 11.5]]
>>> # Print out the type of house
>>>
>>> print(type(house))
<class 'list'>
```

```
# Example
# area variables (in square meters)
hall = 12.45
kit = 17.0
liv = 30.0
bed = 13.55
bath = 11.50
# house information as list of lists
house = [["hallway", hall],
         ["kitchen", kit],
         ["living room", liv],
         ["bedroom", bed],
         ["bathroom", bath]]

# Print out house
print(house)
# Print out the type of house
print(type(house))
```

Source: DataCamp; Wes McKinney (2022)

List basics

Subset and calculate

- Getting separate elements of the list and perform some calculations:

```
# Example
x = ["x", "y", "z", "a"]
print(x[0] + x[2])
```

```
>>> x = ["x", "y", "z", "a"]
>>> print(x[0] + x[2])
xz
```

Example

```
# Example
# Create the areas list
areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0, "bedroom", 13.55, "bathroom", 11.50]
# Sum of kitchen and living area: eat_live_area
eat_live_area = areas[3] + areas[-5]
# Print the variable eat_live_area
print(eat_live_area)
```

```
>>> areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0, "bedroom", 13.55, "bathroom", 11.50]
>>> # Sum of kitchen and living area: eat_live_area
>>>
>>> eat_live_area = areas[3] + areas[-5]
>>> # Print the variable eat_live_area
>>>
>>> print(eat_live_area)
47.0
```

Source: DataCamp; Wes McKinney (2022)

List basics

Slicing and dicing

- You can also perform list slicing, which involves selecting multiple elements from your list. Utilize the following syntax for list slicing (The elements with index 1 and 2 are included, while the element with index 3 is not):

```
# Example
x = ["x", "y", "z", "a"]
x[1:3]
```

```
>>> x = ["x", "y", "z", "a"]
>>> x[1:3]
['y', 'z']
```

Example

```
# Create the areas list
areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0, "bedroom", 13.55, "bathroom", 11.50]
# Use slicing to create downstairs
downstairs = areas[:4]
# Use slicing to create upstairs
upstairs = areas[4:10]
# Print out downstairs and upstairs
print(downstairs)
print(upstairs)
```

```
>>> print(downstairs)
['hallway', 12.45, 'kitchen', 17.0]
>>> print(upstairs)
['living room', 30.0, 'bedroom', 13.55, 'bathroom', 11.5]
```

Source: DataCamp; Wes McKinney (2022)

List basics

Slicing and dicing (2)

- It’s possible not to specify start or end of the index, still using ‘:’.

```
# Example
x = ["x", "y", "z", "a"]
x[:3]
x[3:]
x[:]
```

```
>>> x = ["x", "y", "z", "a"]
>>> x[:3]
['x', 'y', 'z']
>>> x[3:]
['a']
>>> x[:]
['x', 'y', 'z', 'a']
```

Example

```
# Create the areas list
areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0, "bedroom", 13.55, "bathroom", 11.50]
# Alternative slicing to create downstairs
downstairs = areas[:6]
# Alternative slicing to create upstairs
upstairs = areas[6:]
# Print out downstairs and upstairs
print(downstairs)
print(upstairs)
```

```
>>> print(downstairs)
['hallway', 12.45, 'kitchen', 17.0, 'living room', 30.0]
>>> print(upstairs)
['bedroom', 13.55, 'bathroom', 11.5]
```

Source: DataCamp; Wes McKinney (2022)

List basics

Replace list elements

- Simply pick out a subset of the list and assign new values to that subset. You have the flexibility to select individual elements or modify entire slices of the list simultaneously.

```
x = ["x", "y", "z", "a"]
x[1] = "r"
print(x)
x[2:] = ["a", "b"]
print(x)
```

```
>>> x = ["x", "y", "z", "a"]
>>> x[1] = "r"
>>> print(x)
['x', 'r', 'z', 'a']
>>> x[2:] = ["a", "b"]
>>> print(x)
['x', 'r', 'a', 'b']
```

Example

```
# Create the areas list
areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0, "bedroom", 13.55, "bathroom", 11.50]
# Correct the bedroom area
areas[-3] = 13.75
# Change "hallway" to "play zone"
areas[0] = "play zone"
print(areas)
```

```
>>> print(areas)
['play zone', 12.45, 'kitchen', 17.0, 'living room', 30.0, 'bedroom', 13.75, 'bathroom', 11.5]
>>>
```

Source: DataCamp; Wes McKinney (2022)

List basics

Extend a list

- Add new elements to the list:

```
x = ["x", "y", "z", "a"]
y = x + ["b", "c"]
print(y)
```

```
>>> x = ["x", "y", "z", "a"]
>>> y = x + ["b", "c"]
>>> print(y)
['x', 'y', 'z', 'a', 'b', 'c']
```

Example

```
# Create the areas list and make some changes
areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0,
         "bedroom", 13.55, "bathroom", 11.50]
# Add game zone data to areas, new list is areas_1
areas_1 = areas + ["game zone", 10.75]
# Add garage data to areas_1, new list is areas_2
areas_2 = areas_1 + ["garage", 15.45]
print(areas_2)
```

```
>>> areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0,
...          "bedroom", 13.55, "bathroom", 11.50]
>>> # Add game zone data to areas, new list is areas_1
>>>
>>> areas_1 = areas + ["game zone", 10.75]
>>> # Add garage data to areas_1, new list is areas_2
>>>
>>> areas_2 = areas_1 + ["garage", 15.45]
>>> print(areas_2)
['hallway', 12.45, 'kitchen', 17.0, 'living room', 30.0, 'bedroom', 13.55, 'bathroom', 11.5, 'game zone', 10.75, 'garage', 15.45]
```

Source: DataCamp; Wes McKinney (2022)

List basics

Delete list elements

- We can also remove elements from your list with the del statement:

```
x = ["x", "y", "z", "a"]
del(x[3])
print(x)
```

```
>>> x = ["x", "y", "z", "a"]
>>> del(x[3])
>>> print(x)
['x', 'y', 'z']
```

Example

```
# From 'areas_2' we delete 'game zone' data
print(areas_2)
del(areas_2[-4:-2])
print(areas_2)
```

```
>>> print(areas_2)
['hallway', 12.45, 'kitchen', 17.0, 'living room', 30.0, 'bedroom', 13.55, 'bathroom', 11.5, 'game zone', 10.75, 'garage', 15.45]
>>> del(areas_2[-4:-2])
>>> print(areas_2)
['hallway', 12.45, 'kitchen', 17.0, 'living room', 30.0, 'bedroom', 13.55, 'bathroom', 11.5, 'garage', 15.45]
```

Source: DataCamp; Wes McKinney (2022)

List basics

Copy list

- By using [] and ':' operators:

```
# Create list areas
areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0, "bedroom", 13.55, "bathroom", 11.50]
# Create areas_copy
areas_copy = areas[:]
# Change areas_copy
areas_copy[0] = 5.0
# Print areas
print(areas_copy)
```

```
>>> areas = ["hallway", 12.45, "kitchen", 17.0, "living room", 30.0, "bedroom", 13.55, "bathroom", 11.50]
>>> # Create areas_copy
>>>
>>> areas_copy = areas[:]
>>> # Change areas_copy
>>>
>>> areas_copy[0] = 5.0
>>> # Print areas
>>>
>>> print(areas_copy)
[5.0, 12.45, 'kitchen', 17.0, 'living room', 30.0, 'bedroom', 13.55, 'bathroom', 11.5]
```

Source: DataCamp; Wes McKinney (2022)

List basics

Sorting

- You can sort a list in place (without creating a new object) by calling its sort function:

```
x = ["x", "y", "z", "a"]
x.sort()
print(x)
```

```
>>> x = ["x", "y", "z", "a"]
>>> x.sort()
>>> print(x)
['a', 'x', 'y', 'z']
```

- You can sort based on some requirements. Let's sort by the elements lengths:

```
# Example
w = ["saw", "small", "He", "foxes", "six"]
w.sort(key=len)
print(w)
```

```
>>> w = ["saw", "small", "He", "foxes", "six"]
>>> w.sort(key=len)
>>> print(w)
['He', 'saw', 'six', 'small', 'foxes']
```

Source: DataCamp; Wes McKinney (2022)

C. Python Dictionary

Dictionary basics

The ***dictionary*** or ***dict*** is important built-in Python data structure.

A dictionary stores a collection of key-value pairs, where key and value are Python objects.

Each key is associated with a value so that a value can be conveniently retrieved, inserted, modified, or deleted given a particular key.

One approach for creating a dictionary is to use curly braces {} and colons to separate keys and values:

```
# Example
empty_dict = {}
d1 = {"a": "some value", "b": [1, 2, 3, 4]}
print(type(d1))
```

```
>>> empty_dict = {}
>>> d1 = {"a": "some value", "b": [1, 2, 3, 4]}
>>> print(type(d1))
<class 'dict'>
```

Source: Wes McKinney (2022)

Dictionary basics

Operations

- You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:

```
# Example
d1[7] = "an integer"
print(d1)
d1["b"]
```

```
>>> d1[7] = "an integer"
>>> d1
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
>>> d1["b"]
[1, 2, 3, 4]
```

- You can check if a dictionary contains a key using the same syntax used for checking whether a list or tuple contains a value:

```
# Example
"b" in d1
```

```
>>> "b" in d1
True
```

- You can delete values using either the del keyword or the pop method (which simultaneously returns the value and deletes the key):

```
# Example
d1[5] = "some value"
print(d1)
del(d1[5])
print(d1)
d1.pop("b")
d1
```

```
>>> d1[5] = "some value"
>>> print(d1)
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer', 5: 'some value'}
>>> del(d1[5])
>>> print(d1)
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
>>> d1.pop("b")
[1, 2, 3, 4]
>>> d1
{'a': 'some value', 7: 'an integer'}
```

Source: Wes McKinney (2022)

D. Python Set

Set basics

A ***set*** is an unordered collection of unique elements.

A set can be created in two ways: via the ***set*** function or via ***curly*** braces:

```
# Example
set([2, 2, 2, 1, 3, 3])
{2, 2, 2, 1, 3, 3}
```

```
>>> set([2, 2, 2, 1, 3, 3])
{1, 2, 3}
>>> {2, 2, 2, 1, 3, 3}
{1, 2, 3}
```

Source: Wes McKinney (2022)

Set basics

Operations

- Sets support mathematical set operations like union, intersection, difference, and symmetric difference.

Example

```
# Example
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}
a.union(b) # union
a | b # union
a.intersection(b) # intersection
a & b # intersection
```

```
>>> a = {1, 2, 3, 4, 5}
>>> b = {3, 4, 5, 6, 7, 8}
>>> a.union(b) # union
{1, 2, 3, 4, 5, 6, 7, 8}
>>> a | b # union
{1, 2, 3, 4, 5, 6, 7, 8}
>>> a.intersection(b) # intersection
{3, 4, 5}
>>> a & b # intersection
{3, 4, 5}
```

Source: Wes McKinney (2022)

Set basics

Python set operations

Function	Alternative syntax	Description
`a.add(x)`	N/A	Add element `x` to set `a`
`a.clear()`	N/A	Reset set `a` to an empty state, discarding all of its elements
`a.remove(x)`	N/A	Remove element `x` from set `a`
`a.pop()`	N/A	Remove an arbitrary element from set `a`, raising `KeyError` if the set is empty
`a.union(b)`	`a | b`	All of the unique elements in `a` and `b`
`a.update(b)`	`a |= b`	Set the contents of `a` to be the union of the elements in `a` and `b`
`a.intersection(b)`	`a & b`	All of the elements in *both* `a` and `b`
`a.intersection_update(b)`	`a &= b`	Set the contents of `a` to be the intersection of the elements in `a` and `b`
`a.difference(b)`	`a - b`	The elements in `a` that are not in `b`
`a.difference_update(b)`	`a -= b`	Set `a` to the elements in `a` that are not in `b`
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in either `a` or `b` but *not both*
`a.symmetric_difference_update(b)`	`a ^= b`	Set `a` to contain the elements in either `a` or `b` but *not both*
`a.issubset(b)`	`<=`	`True` if the elements of `a` are all contained in `b`
`a.issuperset(b)`	`>=`	`True` if the elements of `b` are all contained in `a`
`a.isdisjoint(b)`	N/A	`True` if `a` and `b` have no elements in common

Source: Wes McKinney (2022)

2. Python Functions

Functions basics

- Functions are the primary and most important method of code organization and reuse in Python.
- As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function.
- Functions can also help make your code more readable by giving a name to a group of Python statements.

Source: Wes McKinney (2022)

Functions basics

In-built functions

- Python comes equipped with a variety of built-in functions designed to simplify the tasks of a data scientist.
- You're already familiar with a couple of these functions: print() and type().
- Additionally, you've employed functions like str(), int(), bool(), and float() to facilitate the conversion between different data types. These functions are also part of Python's built-in arsenal.
- The general way for calling functions and saving the result to a variable is: *output = function_name(input)*

Source: Wes McKinney (2022)

Functions basics

Example

```
# Create variables v1 and v2
v1 = [10, 20, 30, 40]
v2 = True
# Print out type of v1
print(type(v1))
# Print out length of v1
print(len(v1))
# Convert v2 to an integer: out2
out2 = int(v2)
```

```
>>> v1 = [10, 20, 30, 40]
>>> v2 = True
>>> # Print out type of v1
>>>
>>> print(type(v1))
<class 'list'>
>>> # Print out length of v1
>>>
>>> print(len(v1))
4
>>> # Convert v2 to an integer: out2
>>>
>>> out2 = int(v2)
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

Help function

- To find details about any function you may use help function:

```
# Example
help(max)
```

```
>>> help(max)
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

Multiple arguments

- A function may have different attributes/arguments.
- Take a moment to review the documentation for the sorted() function by entering help(sorted) in your Python environment.
- You will observe that sorted() accepts three parameters: iterable, key, and reverse.
- key=None means that if you don't specify the key argument, it will be None.
- reverse=False means that if you don't specify the reverse argument, it will be False, by default.

Source: DataCamp; Wes McKinney (2022)

Functions basics

Multiple arguments

- Example:

```
# Create lists first and second
first = [14.55, 23.0, 5.0]
second = [17.30, 4.75]
# Paste together first and second: full
full = first + second
# Sort full in descending order: full_sorted
full_sorted = sorted(full, reverse = True)
# Print out full_sorted
print(full_sorted)
```

```
>>> first = [14.55, 23.0, 5.0]
>>> second = [17.30, 4.75]
>>> # Paste together first and second: full
>>>
>>> full = first + second
>>> # Sort full in descending order: full_sorted
>>>
>>> full_sorted = sorted(full, reverse = True)
>>> # Print out full_sorted
>>>
>>> print(full_sorted)
[23.0, 17.3, 14.55, 5.0, 4.75]
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

String Methods

- Strings are equipped with a variety of methods.

Example

```
# string to experiment with: place
place = "game zone"
# Use upper() on place: place_up
place_up = place.upper()
# Print out place and place_up
print(place); print(place_up)
# Print out the number of o's in place
print(place.count("o"))
```

```
>>> place = "game zone"
>>> # Use upper() on place: place_up
>>>
>>> place_up = place.upper()
>>> # Print out place and place_up
>>>
>>>
>>> print(place); print(place_up)
game zone
GAME ZONE
>>> # Print out the number of o's in place
>>>
>>> print(place.count("o"))
1
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

List Methods

- Strings aren't the sole Python types that feature associated methods. Lists, floats, integers, and booleans are also data types that include a plethora of helpful methods. In this exercise, you'll have the opportunity to explore:

Example

```
# Create list areas
areas = [14.55, 23.0, 5.0, 17.30, 4.75]
# Print out the index of the element 5.0
print(areas.index(5.0))
# Print out how often 5.0 appears in areas
print(areas.count( 5.0))
# Use append twice to add poolhouse and garage size
areas.append(34.25)
areas.append(19.75)
# Print out areas
print(areas)
# Reverse the orders of the elements in areas
areas.reverse()
# Print out areas
print(areas)
```

```
>>> areas = [14.55, 23.0, 5.0, 17.30, 4.75]
>>> # Print out the index of the element 5.0
>>>
>>> print(areas.index(5.0))
2
>>> # Print out how often 5.0 appears in areas
>>>
>>> print(areas.count( 5.0))
1
>>> # Use append twice to add poolhouse and garage size
>>>
>>> areas.append(34.25)
>>> areas.append(19.75)
>>> # Print out areas
>>>
>>> print(areas)
[14.55, 23.0, 5.0, 17.3, 4.75, 34.25, 19.75]
>>> # Reverse the orders of the elements in areas
>>>
>>> areas.reverse()
>>> # Print out areas
>>>
>>> print(areas)
[19.75, 34.25, 4.75, 17.3, 5.0, 23.0, 14.55]
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

Writing a function

- Sometimes it’s more efficient to write own function.

Example

```
# Example
def my_function(x, y):
    return x + y

my_function(5, 7)
```

```
>>> def my_function(x, y):
...     return x + y
...
>>> my_function(5, 7)
12
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

Import package

- You can get much more functions from some specific packages.

Example

```
# Definition of radius
r = 0.87
# Import the math package
import math
# Calculate C (the circumference of the circle)
C = 2 * math.pi * r # pi is fixed number ~3.14
# Calculate A (the area of the circle)
A = math.pi * (r**2) # '**' power operator
# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```

```
>>> r = 0.87
>>> # Import the math package
>>>
>>> import math
>>> # Calculate C (the circumference of the circle)
>>>
>>> C = 2 * math.pi * r # pi is fixed number ~3.14
>>> # Calculate A (the area of the circle)
>>>
>>> A = math.pi * (r**2) # '**' power operator
>>> # Build printout
>>>
>>> print("Circumference: " + str(C))
Circumference: 5.46637121724624
>>> print("Area: " + str(A))
Area: 2.3778714795021143
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

Selective import from package

- When you make general imports, such as import math, you gain access to all the functionality provided by the entire math package. However, if you prefer to use only a particular component of a package, you have the option to make your import more specific: ‘from math import pi’

Example

```
# Definition of radius
r = 0.87
# Import pi function of math package
from math import pi
# Calculate C (the circumference of the circle)
C = 2 * pi * r # pi is fixed number ~3.14
# Calculate A (the area of the circle)
A = pi * (r**2) # '**' power operator
# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```

```
>>> r = 0.87
>>> # Import pi function of math package
>>>
>>> from math import pi
>>> # Calculate C (the circumference of the circle)
>>>
>>> C = 2 * pi * r # pi is fixed number ~3.14
>>> # Calculate A (the area of the circle)
>>>
>>> A = pi * (r**2) # '**' power operator
>>> # Build printout
>>>
>>> print("Circumference: " + str(C))
Circumference: 5.46637121724624
>>> print("Area: " + str(A))
Area: 2.3778714795021143
```

Source: DataCamp; Wes McKinney (2022)

Functions basics

Different ways of importing

- There exist multiple approaches to importing packages and modules into Python. Depending on the specific import statement you choose, you will need to employ different Python code.

Example

- Let's say you want to use the inv() function, which is located in the *linalg* subpackage of the *scipy* package, then to get it you run:

```
from scipy.linalg import inv as my_inv
```

Source: DataCamp; Wes McKinney (2022)

3. Files and the Operating System

Files basics

It’s important to understand the basics of how to work with files in Python.

Fortunately, it’s relatively *straightforward*, which is one reason Python is so popular for text and file munging.

- To open a file for reading or writing, use the built-in open function with either a relative or absolute file path and an optional file encoding.
- We pass encoding="utf-8" as a best practice because the default Unicode encoding for reading files varies from platform to platform.
- By default, the file is opened in read-only mode "r".

Source: Wes McKinney (2022)

Files basics

Example

- We can then treat the file object f like a list and iterate over the lines like so:

```
# Example
path = "C:/My docs/VScode_traning/MPS.txt"
f = open(path, encoding="utf-8")
for line in f:
    print(line)
f.close()
```

```
>>> path = "C:/My docs/VScode_traning/MPS.txt"
>>> f = open(path, encoding="utf-8")
>>> for line in f:
...     print(line)
...
NBU Cuts Key Policy Rate to 22%

 27 Jul 2023 14:00

The Board of the National Bank of Ukraine has decide
tions enabled the central bank to start the cycle of
n, the current and forecast rate cuts conform to mai
 amid the easing of FX restrictions and shifting to
financial stability will support economic recovery.
```

Source: Wes McKinney (2022)

Files basics

Important Python file methods or attributes

Method/attribute	Description
`read([size])`	Return data from file as bytes or string depending on the file mode, with optional `size` argument indicating the number of bytes or string characters to read
`readable()`	Return `True` if the file supports `read` operations
`readlines([size])`	Return list of lines in the file, with optional `size` argument
`write(string)`	Write passed string to file
`writable()`	Return `True` if the file supports `write` operations
`writelines(strings)`	Write passed sequence of strings to the file
`close()`	Close the file object
`flush()`	Flush the internal I/O buffer to disk
`seek(pos)`	Move to indicated file position (integer)
`seekable()`	Return `True` if the file object supports seeking and thus random access (some file-like objects do not)
`tell()`	Return current file position as integer
`closed`	`True` if the file is closed
`encoding`	The encoding used to interpret bytes in the file as Unicode (typically UTF-8)

Source: Wes McKinney (2022)

4. In-class Assignment