**Python Collection Data Types** 1

Overview

In this handout, we will cover common collection data types. This includes sequence data types such as list and tuple and we will expand our knowledge of range, and string data types. Additionally, we will learn about non-sequence collection data types such as diet, set, and frozenset. Understanding these data structures will enable you to efficiently store, retrieve, and manage data. Additionally, we will learn about the less-known concept of generator expressions; which allow for efficient iteration over large datasets without the need to load everything into memory all at once.

Understanding these data types will provide a necessary foundation for working with pandas data structures later on.

## 1 Introduction

In programming, we often need to store and manage multiple values together rather than individual pieces of data. Python provides a variety of **collection data types** that allow us to store more than one data point efficiently. These data types help us manage and organize data, making it easier to manipulate and access information as needed.

Python's collection data types can be broadly classified into two categories:

* **Sequence Data Types:** These store elements in a specific order, allowing access to elements using an index. Common sequence types include:
  + str: A sequence of characters. We have learned about str in the previous handout.
  + range: A sequence representing a range of numbers. We have learned about range in the previous handout.
  + list: A mutable sequence containing elements of any type.
  + tuple: An immutable sequence containing elements of any type.
* **Non-Sequence Data Types:** These store data but do not maintain a specific order of elements, and thus, cannot be accessed using an index. Common non-sequence types include:
  + diet: A collection of key-value pairs, allowing quick access to values through their keys.
  + set: An unordered collection of unique elements.

Understanding the differences between these types helps us select the right data structure for the task at hand, improving the efficiency and clarity of our code.

1These notes and examples adapt the references listed at the end. They are compiled to fit the scope of this specific course.

Collection, Sequence, and Non-Sequence Data Types

list 1

list 2

list 3

list 4

[]

[5, 10, 15, 20, 30, 100]

# *an empty list*

# *homogeneous data*

['python', 'Java', 'Data Science', 'I feel good'] # *homogeneous data*

["I'm in love with ", 'python ', 3.12]

# *heterogeneous data*

**2 Sequence Data Types: list**

A list is a data structure in Python that can store multiple items. These items are separated by commas , and enclosed within square brackets [J. While lists often store items of the same type (homogeneous), they can also contain items of different types (heterogeneous). Here are some examples:

**Examples of a** list

**collection data types** store multiple data points in a single structure. These types are categorized into:

* **Sequence Data Types:** Maintain a specific order and allow indexed access to elements, like str, list, tuple, and range.
* **Non-Sequence Data Types:** Do not maintain a specific order, and elements cannot be accessed by index, such as set and diet.
  1. **Built-in** lenO, type(), max(), min(), **and** sum() **Functions with a** list

Python provides several built-in functions that make working with lists2 easy and efficient. Let's explore some of these:

* + - len() returns the number of items in a list.
    - type O tells you the data type of an object.
    - max O finds the largest value in a list.
    - min() finds the smallest value in a list.
    - sum() calculates the total of all numbers in a list.

Let's see an example:

**Using built-in functions with a** list temperatures= [72, 68, 75, 70, 69, 64, 61]

num\_days = len(temperatures) print(f'Number of days: {num\_days}')

temp\_type = type(temperatures) print(f'Type of object: {temp\_type}')

2These built-in functions work with other collection data types such as tuple, diet, set, andrange.

max\_temp = max(temperatures) print(f'Highest temperature: {max\_temp}')

min\_temp = min(temperatures) print(f'Lowest temperature: {min\_temp}')

total\_temp = sum(temperatures) print(f'Total temperature: {total\_temp}')

average\_temp = total\_temp / num\_days print(f'Average temperature: {average\_temp:.2f}')

This code gives the following output:

Number of days: 7

Type of object: <class 'list'> Highest temperature: 75

Lowest temperature: 61

Total temperature: 479

Average temperature: 68.43

List

**2.2 Accessing** list **Elements with Indices (Indexes)**

A list is an ***iterable,*** meaning its elements can be accessed one by one. Each element in a list is associated with an **index,** which is used to retrieve the element. Indexing in a list starts at o, so the first element is at index o, the second at index 1, and so forth, up to len(list\_name) - 1 for the last element.

In addition to positive indices, you can also use **negative indices** to access elements from the end of the list. The index -1 refers to the last element, -2 to the second-to-last, and so on, up to -len(list\_name) for the first element.

Here is an example:

**Accessing** list **elements using positive and negative indices**

days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

print(f'The first day is: {days[0]}') print(f'The third day is: {days[2]}')

print(f'The last day is: {days[-1]}')

A **list** is an ordered and mutable collection in Python, allowing access to elements by index. Lists can store multiple items of different data types.

print(f'The second-to-last day is: {days[-2]}')

This code produces:

The first day is: Monday The third day is: Wednesday The last day is: Sunday

The second-to-last day is: Saturday

* 1. **Common Error:** IndexError: list index out of range

It is normal -and expected- to encounter index-related errors, especially if you are new to working with 0-based indices. Lists start counting from o, which means the first element is at index o, not 1. This can be confusing at first, often leading to mistakes when accessing list elements. If you try to access an index that doesn't exist in the list, you will get the IndexError.

Here is an example:

**accessing list elements using positive and negative indices**

days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

print(f'The last day is: {days[7]}')

produces

**IndexError:** list index out of range

because there is no item at index 7. The last element has an index of 6.

#### Mutability of a list

A list is **mutable,** which means its elements can be modified *after* the list has been created. In simpler terms, lists are dynamic data structures-elements can be added, removed, or changed without needing to create a new list. This flexibility makes lists highly useful in situations where you need to update data frequently.

For example, consider the following code, where we modify the second element of a list:

**modifying a** list **element**

grades= [88, 92, 75, 89, 94]

grades[2] = 85 # *correct the 3rd grade*

print(f'updated grades is: {grades}')

This will produce the following output:

updated grades is: [88, 92, 85, 89, 94]

This ability to change list elements is what distinguishes lists from tuples, which we will discuss later.

* 1. **List Packing Using** append() **Method**

A list can be created by specifying its elements directly, like in the examples above. Alternatively, you can start with an empty list [J and gradually add elements using the append O method3. append O adds one element to the end of the list each time it is used.

**Using** append() **to Build a** list

student\_scores = []

# *empty list*

student\_scores.append(95) # *Adding a new test score*

print(student\_scores)

This code will output:

[95]

**List Packing:** In the above example, append() effectively *packs* the element 95 into the list student\_scores. List packing is the process of adding elements to a list, allowing them to be stored together in a single data structure. Each time you use append(), you add a new element to the packed list.

Let's continue adding more elements:

**Adding more items using** append()

student\_scores.append('Extra Credit') # *Adding a note*

student\_scores.append(True) print(student\_scores)

This results in:

[95, 'Extra Credit', True]

It is a common practice to start with an empty list and add elements using a for loop. For instance, let's build a list of even numbers up to 10:

**Using** append() **inside a loop**

|  |  |
| --- | --- |
| even\_numbers = [] | # *start with an empty list* |
| for num in range(2, 11, 2): | # *start with 2 and with step size of 2* |
| even\_numbers.append(num) | # *use append inside a for loop* |
| print(even\_numbers) | # *print the results when done* |

This will produce the following output:

3A method is a function that is specific to a particular data type.

[2, 4, 6, 8, 10]

Here, the for loop repeatedly packs each even number into the even\_numbers list using append() method. This allows you to build and expand lists easily, adding elements one at a time.

#### List Packing Using += Operator

An alternative to the append() method is using+= to add elements to a list. However, use it cautiously, as it can produce unexpected results or errors!

**Incorrect Use of** +=

grades= [] # *start with an empty list*

grades+= 80 # *attempt to add 80 to the list using+=*

This results in a

TypeError: 'int' object is not iterable Here's why:

When using+= with a list, the right side must be an *iterable-* an object from which items can be retrieved

one by one. In this example, 80 is an int, not an iterable, causing the error. To add 80 using+=, wrap it in square brackets [J, making it a list:

**r-iii4M1HI**

grades = []

grades += [80]

**Now,** grades becomes [80].

Here's how to create a list of non-negative integers less than 10 using +=:

**Building a** list **Using** +=

my\_numbers = []

for item in range(10): my\_numbers += [item]

print(my\_numbers)

# *empty list*

# *Wrap each item in a list before adding*

yields

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Remember that as an alternative, you could use append O as below:

**Using** append() **to build a** list

my\_numbers = []

**for** item **in** range(10): my\_numbers.append(item)

print(my\_numbers)

yielding the same list.

Be cautious when using += to add strings to a list. Here's an example that shows the difference between using += and append():

**Comparing+= and** append() **with** str

my \_list\_1 = [] my\_list\_1 += 'python'

print(f'0utput using+=: {my\_list\_1}')

# *empty list*

# *Output: ['p', 'y', 't', 'h',*

*101,*

*'n']*

my \_list\_2 = [] my\_list\_2.append('python')

print(f'0utput using append(): {my\_list\_2}')

# *empty list*

# *Output: ['python']*

This code yields:

Using +=: ['p', 'y', 't', 'h', 'o', 'n'] Using append(): ['python']

In the first case, my \_list\_1 += 'python' adds each character of the string 'python' as a separate element

in the list. This results in ['p', 'y', 't', 'h', 'o', 'n']. lnthesecondcase,my\_list\_2.append('python') adds the entire string 'python' as a single element, resulting in ['python' J .

* 1. **Creating a** list **Using the Built-in** list O **Function**

In addition to the append O method and the += operator, the built-in list O function provides another way to create lists. This function is particularly useful for converting various types of iterables - such as strings, ranges, and tuples4 - into a list.

Here are some examples of using list O:

**Converting a** string **into a** list O

word= "Rock & Roll" chars= list(word) print(chars)

This code converts the string "Rock & Roll" into a list of its characters:

4To be discussed later in this handout

['R', 'o', 'c', 'k',' ', '&',' ', 'R', 'o', 'l', 'l']

The list() function is also useful for converting range into a list:

**Converting a** range **into a** list

number\_range = range(5) number\_list = list(number\_range) print(number\_list)

This code converts a range of numbers into a list producing:

[0, 1, 2, 3, 4]

Later on, we will see that we can use list() function with other iterables such as a tuple, dictionary and a set.



The list O function is used to create a new list. It can convert an iterable into a list.

#### List Creation Using List Comprehension

So far, we have seen how to create a list using the append() method, the+= operator, and the list O function. Another powerful and concise way to create a new list is by using **list comprehension.** List comprehension allows us to create a new list from an existing sequence in just one line, making the code shorter and cleaner.

Let's start with an example that does not use list comprehension:

**Without** list **Comprehension**

initial\_list = [2, 5, 7, 19, 20, 41, 50] even\_nums = []

for item in initial\_list: if item% 2 == 0:

even\_nums.append(item) print(even\_nums)

# *initial List*

# *an empty List*

# *check even-ness*

This code creates an empty list,and through a loop and a conditional statement, adds even numbers from

initial\_list to even\_numbers. The output is:

[2, 20, 50]

With list comprehension, we can write this in a much cleaner way by bringing the for and if inside the square brackets [ J:

**With** list **Comprehension**

initial\_list = [2, 5, 7, 19, 20, 41, 50]

even\_nums = [item **for** item **in** initial list if item% 2 print(even\_nums)

# *initial list*

OJ # *list comprehension*

This one line of code does the same job as the previous code, but it's easier to read and understand. The output is:

[2, 20, 50]

The general syntax for list comprehension is:

new\_list = [expression for itern in iterable if condition== True]

Notice that the new list is enclosed inside a pair of square brackets [ J .

Let's see more examples:

list **Comprehension**

list\_1 = [2, 3, 7]

list\_1\_squared = [item\*\* 2 for item in list 1] print(list\_1\_squared)

# *squares of numbers*

list\_2 = list(range(1, 6))

list 2 inverted= [round(1 / item, 2) for item in list 2] # *rounded 1/numbers*

print(list\_2\_inverted)

before\_tax = [20, 50, 120]

tax\_rate = 0.08

after\_tax = [round(item \* (1 + tax\_rate), 2) for item in before tax] # *after tax*

print(after\_tax)

names= ['Milo', 'Anna', 'Python']

names\_up = [\_.upper() **for in** names **if** 'o' **in** ] # *uppercase names with 'o'*

print(names\_up)

The code above creates new lists using list comprehension for various scenarios. The output is:

[4, 9, 49]

[1.0, 0.5, 0.33, 0.25, 0.2]

[21.6, 54.0, 129.6]

['MILO', 'PYTHON']

remember that is a valid identifier.

List comprehension provides a simple and elegant way to create new lists, making the code more readable.

#### Notable list methods (functions) and operators

A list is mutable, meaning it can be changed after it is created. This flexibility comes with various methods that let us perform various operations. The table below highlights the most important methods, followed by examples to show their use.

|  |  |  |
| --- | --- | --- |
| **Method** | **Sample Syntax** | **Description** |
| append() | list.append(item) | Adds itern to the end of the list. |
| insert() | list.insert(index, item) | Inserts itern at the specified index.  Shifts subsequent elements. |
| extend() | list.extend(iterable) | Adds all elements from iterable to the end of the list. |
| count() | list.count(item) | Counts the number of occurrences of itern in the list. |
| index() | list.index(item) | Returns the index of the first occurrence of itern.  Raises ValueError if not found. |
| remove() | list.remove(item) | Removes the first occurrence of itern.  Raises ValueError if not found. |
| pop() | list.pop(index) | Removes and returns the item at index.  Defaults to the last item if index is omitted. |
| sort() | list.sort() | Sorts the list in ascending order.  Use list.sort(reverse=True) for descending. |
| reverse() | list.reverse() | Reverses the order of items in the list, in place. |
| copy() | list.copy() | Returns a shallow·copy of the list. |
| clear() | list.clear() | Removes all elements from the list. |

* + - *As opposed to a deep copy More on this later.*

Table 1: Important List Methods



my\_list = [3, 5, 7, 9, 3] print(f'initial list: {my\_list}')

my\_list.append(11)

print(f'after append 11: {my\_list}') # *[3, 5,* 7, *9, 3, 11]*

my\_list.insert(2, 4)

print(f'after insert 4 at index 2: {my\_list}') # *[3, 5, 4,* 7, *9, 3, 11]*

my\_list.extend([13, 15])

print(f'after extend [13, 15]: {my\_list}') # *[3, 5, 4,* 7, *9, 3, 11, 13, 15]*

count\_of\_3 = my\_list.count(3)

print(f'count of 3 in {my\_list}:{count\_of\_3}') # *2*

index\_of\_7 = my\_list.index(7)

print(f'index of first 7 in {my\_list}: {index\_of\_7}') # *3*

my\_list.remove(3)

print(f'after removal of first 3: {my\_list}') # *[5, 4,* 7, *9, 3, 11, 13, 15]*

last\_item = my\_list.pop()

print(f'after pop: {my\_list}, Popped item: {last\_item}') # *[5, 4,* 7, *9, 3, 11, 13],*

'-+ *Popped item: 15*

my\_list.sort()

print(f'after sort: {my\_list}') # *[3, 4, 5,* 7, *9, 11, 13]*

my\_list.reverse()

print(f'after reverse: {my\_list}') # *[13, 11, 9,* 7, *5, 4, 3]*

copied\_list = my\_list.copy()

print(f'copied list: {copied\_list}') # *[13, 11, 9,* 7, *5, 4, 3]*

my\_list.clear()

print(f'after clear: {my\_list}') # *[]*

initial list: [3' 5, 7, 9, 3]

after append 11: [3' 5, 7, 9, 3, 11]

after insert 4 at index 2: [3' 5, 4, 7, 9, 3, 11]

after extend [13, 15]: [3' 5, 4, 7, 9, 3, 11, 13,

count of 3 in [3, 5, 4, 7, 9, 3, 11, 13, 15]: 2

index of first 7 in [3' 5, 4, 7, 9, 3, 11, 13, 15]:

after removal of first 3: [5' 4, 7, 9, 3, 11, 13,

15]

3

15]

after pop: [5, 4, 7, 9, 3, 11, 13], Popped itern: 15

after sort: [3, 4, 5, 7, 9, 11, 13]

after reverse: [13, 11, 9, 7, 5, 4, 3]

copied list: [13, 11, 9, 7, 5, 4, 3] after clear: []

###### Besides the above methods, we often use the following operators with a list.

|  |  |  |
| --- | --- | --- |
| **Operator/Keyword** | **Sample Syntax** | **Description** |
| in | i tern in list | Checks if i tern exists in the list.  Returns True if it exists. |
| not in | i tern not in list | Checks if i tern does not exist in the list.  Returns True if it does not exist. |
| -- | list1 == list2 | Returns True if list1 and list2 are equal. |
| != | list1 !=list2 | Returns True if list1 and list2 are not equal. |
| + | list1 + list2 | Concatenates list1 and list2 into a new list. |
| \* | list\* n | Repeats list ntimes and returns a new list. |
| del | del list [index]  del list | Deletes the element at index from the list.  Can also delete the entire list. |

Table 2: Operators and keywords with list

**Examples of List Operators and Keywords**

list1 list2

[1, 2, 3]

[4, 5, 6]

print(f"Is 2 in list1? {'Yes' if 2 in list1 else 'No'}") print(f"Is 7 in list1? {'Yes' **if** 7 **in** list1 **else** 'No'}")

# *Yes*

# *No*

print(f"Is 7 not in list1? {'Yes' if 7 **not in** list1 **else** 'No'}") # *Yes*

print(f"Is 2 not in list1? {'Yes' if 2 **not in** list1 **else** 'No'}") # *No*

print(f"Is list1 equal to [1, **2,** 3]? {list1 -- [1, **2,** 3]}") # *True*

print(f"Is list1 equal to list2? {list1 == list2}") # *FaLse*

print(f"Is list1 different from list2? {list1 != list2}") # *True*

print(f"Is list1 different from [1, 2, 3]? {list1 != [1, 2, 3]}") # *FaLse*

new\_list = list1 + list2 # *'+' for Concatenation (Joining)*

print(f"Concatenated list1 andlist2: {new\_list}") # *[1, 2, 3, 4, 5, 6]*

repeated\_list = list1 \* 2

print(f"Repeating list1 twice: {repeated\_list}")

# *'\*' for Repetition*

# *[1, 2, 3, 1, 2, 3]*

**del** list1[1] # *deL keyword for deLeting one item*

print(f"After deleting index 1 from list1: {list1}") # *[1, 3]*

del list1 # *del keyword for deleting the entire list*

Is 2 in list1? Yes Is 7 in list1? No

Is 7 not in list1? Yes Is 2 not in list1? No

Is list1 equal to [1, 2, 3]? True Is list1 equal to list2? False

Is list1 different from list2? True

Is list1 different from [1, 2, 3]? False

Concatenated list1 andlist2: [1, 2, 3, 4, 5, 6]

Repeating list1 twice: [1, 2, 3, 1, 2, 3]

After deleting index 1 from list1: [1, 3]

* 1. **Difference Between** clear() **and** del

The clear O method and the del keyword are both used to remove elements, but they work differently.

* + - clear():
      * This method is specific to lists 5
      * It removes all elements from a list, leaving it empty, but the list itself still exists and can be used. 6

**Using** clear O **Method on a** list

my\_list = [1, 2, 3] my \_list. clear() print(my\_list)

yields the empty list of [ J. After using clear O, my \_list isstill a valid, empty list.

* + - del:
      * delis a Python keyword, not limited to lists.7
      * It can be used to delete variables, list elements or parts (slices) of lists.
      * When used with a list, del can remove specific elements or it can delete the entire list object.

**Using** del **on a** list

]

# *Removes the element at index 1 (value 2)*

# *Output: [1, 3]*

my\_list = [1, 2, 3 del my\_list[1] print(my\_list)

5and some other data structures like dictionaries, covered later

6In other words, the list object itself remains

7We have used it in the previous handout with a string

del my\_list # *Deletes the entire list object*

print(my\_list) # *NameError as my\_list no longer exists*

This yields

[1, 3]

followed by

NameError: name 'my\_list' is not defined

* 1. sort() **Method Versus the Built-in** sorted() **Function**

The sort() method and the sorted() function are both used to sort a list; but:

* + - sort O is a list method that modifies the original list **in-place8.** This means that it does not return a new list but directly changes the order of elements in the list itself. Once sorted, the original order is lost.
    - sorted(), on the other hand, is a built-in function that returns a new list with the elements sorted, leaving the original list unchanged. It works with any iterable, not just lists, and can be used when you need to preserve the original sequence.

Here are examples displaying these differences:

**Using** sort() **Method**

grades= [85, 72, 90, 88, 78]

grades.sort()

print(f'After using sort(): {grades}') # *The original grades list is now sorted*

This yields:

After using sort(): [72, 78, 85, 88, 90]

**Using** sorted() **Built-in Function**

grades= [85, 72, 90, 88, 78]

sorted\_grades = sorted(grades)

print(f'Driginal list: {grades}')

# *The original list remains unchanged*

print(f'Sorted list: {sorted\_grades}') *#Anew list is created with sorted values*

This yields:

Original list: [85, 72, 90, 88, 78]

Sorted list: [72, 78, 85, 88, 90]

8You'II hear this term often, especially when we discuss pandas. "In-place" means that it modifies the original object.

Both sort() and sorted() accept optional parameters such asreverse=True to sort the contents in descending order. 9

sort() **vs** sorted()

**sort()** is a list method that sorts the list in place, modifying the original list. **sorted()** is a function that returns a new sorted list from any iterable without changing the original.

##### Two-Dimensional Lists

A standard list stores data in one dimension, but for many practical tasks, we need to handle two-dimensional data such as matrices or tables. A two-dimensional list is a *list of lists,* where each sublist represents a *row.*

##### Creating a Two-Dimensional List

###### Two-Dimensional List

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| two\_dim\_list = [[1, print(two\_dim\_list) | 2, | 3], | [4, | 5, | 6], | [7, | 8]] |

This prints:

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

##### Accessing Rows

You can loop over the rows of a two-dimensional list like this:

**r,wifii+**

■**,**



**for r in two\_dim\_list,**

\_ print(r)

This prints

|  |  |
| --- | --- |
| [1, | 2, 3] |
| [4, | 5, 6] |
| [7, | 8] |

##### Accessing Columns

You can access a column by using a simple loop to extract the element from each row:

9You can also define a custom key function for more complex sorting criteria. More on this later.

**Accessing Columns**

column\_1 = []

for row in two dim\_list:

column\_1.append(row[O]) # *Append the first element of each row*

print(column\_1)

This prints:

[1, 4, 7]

This method iterates through each row, appending the first element to the column\_1 list.

##### Accessing Elements

To access individual elements, we use two indices: one for the row and one for the column:

**Accessing Elements**

two\_dim\_list = [[1, 2, 3], [4, 5, 6], [7, 8]]

print(f'Element at 1st row and 2nd column: {two dim\_list[O][1]}') print(f'Element at 3rd row and 2nd column: {two\_dim\_list[2][1]}')

This prints:

Element at 1st row and 2nd column: 2 Element at 3rd row and 2nd column: 8

Later in this handout, we will explore other ways to store two-dimensional data, such as using *lists of tuples* and *lists of dictionaries.* Additionally, in the following handouts, we will cover more efficient methods for handling multi-dimensional data using numpy.

## Sequence Data Types: tuple

A tuple consists of values separated by commas , and is enclosed in (optional) parentheses O. Here are a few examples:



empty\_playlist = ()

# *empty tuple*

playlist\_1 'Rock', 'Jazz', 'Pop' *#()are optional but recommended*

playlist\_2 ('Rock' , 'Jazz' , 'Pop')

solo\_performance\_duration = (20,)

# *needs a comma to be recognized as a tuple*

print(f'Number of songs in empty\_playlist: {len(empty\_playlist)}') print(f'playlist\_1: {playlist\_1}')

print(f'playlist\_1 == playlist\_2?: {playlist\_1 == playlist\_2}') *#is() optional?*

print(f'The second item in playlist\_2 is: {playlist\_2[1]}')

print(f'solo\_performance\_duration: {solo\_performance\_duration}')

This yields:

Number of songs in empty\_playlist: 0 playlist\_1: ('Rock', 'Jazz', 'Pop') paylist\_1 == playlist\_2?: True

The second item in playlist\_2 is: Jazz solo\_performance\_duration: (20,)

As you can see, the parentheses O are optional, but including them is recommended for better code read­ ability. When defining a tuple with a single element, like solo\_performance\_duration (a **singleton tuple),** the comma at the end signals Python that this is a tuple. Without the comma, solo\_performance\_duration would be interpreted as an int.

uple

A **tuple** is an ordered and immutable collection. Once created, its elements cannot be modified, and items can be accessed by index.

#### Accessing and Modifying Elements

Accessing elements of a tuple is similar to accessing elements of a list-you use zero-based indexing with the index placed inside square brackets [J. Once a tuple is defined, you **cannot** change its elements; you cannot assign a new value to any specific element. Similar to the built-in str type, tuple is also **immutable.** This means that any attempt to change an element in a tuple will result in an error.

Here is an example:

**Immutability of a** string

str\_1 = 'Care'

str\_1[1] = 'o' # *try to replace 'a' with an 'o'*

This yields:

TypeError: 'str' object does not support item assignment

This is the same error you get if you try to modify an element of a tuple. Here is an example:

**Immutability of a** tuple

tuple\_a = (10, 20, 30, 40)

tuple\_a[2] = 25

This yields:

TypeError: 'tuple' object does not support item assignment

* 1. **Using+= with a** tuple

Item assignment using square brackets [J is different from using += to add elements to a tuple. Using the augmented assignment of += creates a new tuple.

Here is an example:

**Using += with Tuples**

tuple\_b = (10, 20, 30)

print(f'old id(tuple\_b): {id(tuple\_b)}')

tuple\_c = (40, 50)

tuple\_b += tuple\_c # *this defines a new tuple*

print(f'new id(tuple\_b): {id(tuple\_b)}')

This yields:

old id(tuple\_b): 2597852246144

new id(tuple\_b): 2597852645632

The id(tuple\_b) is different before and after using+=, which means Python has created a new tuple\_b. **A bit about the id function:** id() is a built-in function that returns a unique identifier for an object. id(my \_object) returns an integer that remains constant for the object during its lifetime. This number can be considered a refer­ ence to the object's location in memory. The specific value of id() may vary between different Python sessions, as objects are stored in different locations in memory each time a session is started.

#### Notable Methods for a tuple

The following table outlines the most commonly used methods for tuple:

|  |  |  |
| --- | --- | --- |
| **Method** | **Sample Syntax** | **Description** |
| count() | tuple\_1.count(x) | Returns the number of times x appears in tuple\_1. |
| index() | tuple\_1.index(value) | Returns the first index (position) of value in tuple\_1.  Raises ValueError if value is not found. |

Table 3: Important Tuple Methods

**Using** count() **and** index O **with a** tuple

scores= (85, 90, 78, 85, 92) # *a tuple*

count\_85 scores.count(85)

print(f'85 appears {count\_85} times in scores: {scores}')

index\_78 = scores.index(78)

print(f'The first occurrence of 78 is at index {index 78}')

85 appears 2 times in scores: (85, 90, 78, 85, 92) The first occurrence of 78 is at index 2

Tuples have fewer methods compared to lists because they are **immutable** The immutability of tuples makes them simpler and more efficient when only read-only data is needed, while lists offer more flexibility for scenarios where the data may need to change.

Additionally, all the operators we discussed for lists (in, not in, ==, ! =, +, \*) and the del keyword work in a similar way with tuples. I encourage you to explore how these operators function with tuples on your own.

* 1. tuple O **Function**

We have previously seen type conversion functions like int O, float O, str O, and list O to convert values into integers, floats, strings, and lists, respectively. Similarly, Python provides tuple O function to convert any iterable (such as strings, ranges, and even other sequences) into a tuple.

Here's an example:

**Converting a** string **to a** tuple

my\_string = 'JHU'

my\_tuple = tuple(my\_string) # *Convert the string into a tuple*

print(my\_tuple)

This yields:

('J', 'H', 'U')

Both list O and tuple O functions are useful when you need to change the type of an iterable for further opera­ tions. For instance, if you want to multiply a range or str by a number or manipulate its elements, converting it into a list or tuple may be necessary.

tuple O function



**4 Sequence Slicing**

So far, we have seen four types of sequences: str, list, tuple, and range. Slicing is a way to create a subset of any of these sequences, and it works in the same way for all four types. Slicing does not change the original sequence; it creates a new one. A **slice** is a subset of a sequence. The slicing syntax is

sequence\_identifier[start\_index : end\_index : step\_size]

This copies elements from the start\_index (inclusive) to the end\_index (exclusive). The step\_size is optional and by default is equal to 1. The step\_size can be a negative number to reverse the sequence.

Here are a few examples

weekly\_sales = [40, 25, 85, 100, 220, 15, 5] # *sales data from Monday to Sunday*

print(f'weekly\_sales is {weekly\_sales}')

print(f'weekly\_sales[1:4] is {weekly\_sales[1:4]}') # *from index 1 to 3* print(f'weekly\_sales[:3] is {weekly\_sales[:3]}') # *from start to index 2* print(f'weekly\_sales[2:] is {weekly\_sales[2:]}') # *from index 2 to end* print(f'weekly\_sales[:] is {weekly\_sales[:]}') # *a (shallow) copy* print(f'weekly\_sales[::2] is {weekly\_sales[::2]}') # *every second element* print(f'weekly\_sales[::-1] is {weekly\_sales[::-1]}') # *reverse the List*

The **tuple()** function is used to create a tuple. It can convert an iterable into a tuple, which is an immutable sequence.

This yields:

weekly\_sales is [40, 25, 85, 100, 220, 15, 5]

weekly\_sales[1:4] is [25, 85, 100]

weekly\_sales[:3] is [40, 25, 85]

weekly\_sales[2:] is [85, 100, 220, 15, 5]

weekly\_sales[:] is [40, 25, 85, 100, 220, 15, 5]

weekly\_sales[::2] is [40, 85, 220, 5]

weekly\_sales[::-1] is [5, 15, 220, 100, 85, 25, 40]

weekly \_sales[:J creates a *shallow* copy of the list. You can use the weekly \_sales.copy() method, alter­ natively. We will discuss the concept of shallow vs. deep copies later.

As a list is mutable, you can modify parts of a list using slicing notation.

**Modifying a** list **using slicing**

weekly\_sales[1:3] = [2000, 3000] # *change elements at index 1 and 2*

print(weekly\_sales)

This yields:

[40, 2000, 3000, 100, 220, 15, 5]

You can also use slicing to remove elements:

**Removing elements using slicing**

weekly\_sales = [40, 25, 85, 100, 220, 15, 5] weekly\_sales[4:] = [] # *remove elements from index 4 onward* print(weekly\_sales)

This yields:

[40, 25, 85, 100]

Although tuples are immutable, you can still use slicing to extract a subset of their elements:



months ('Jan' 'Feb'' 'Mar'' 'Apr'' 'May'' 'Jun'' 'Jul'' 'Aug')

print(f'months[2:5] is {months[2:5]}')

print(f'months[:3] is {months[:3]}')

print(f'months[::2] is {months[::2]}')

# *elements from index 2 to 4*

# *elements from start to index 2*

# *every second element*

This yields:

months[2:5] is ('Mar', 'Apr', 'May')

months[:3] is ('Jan', 'Feb', 'Mar')

months[::2] is ('Jan', 'Mar', 'May', 'Jul')

Slicing also works with range objects. Here's how:

range **slicing**

my\_range = range(10)

print(f'my\_range[2:6] is {list(my\_range[2:6])}') # *from index 2 to 5* print(f'my\_range[::3] is {list(my\_range[::3])}') # *every third element* print(f'my\_range[::-1] is {list(my\_range[::-1])}') # *reverse the range*

This yields:

my\_range[2:6] is [2, 3, 4, 5]

my\_range[::3] is [0, 3, 6, 9]

my\_range[::-1] is [9, 8, 7, 6, 5, 4, 3, 2, 1, OJ

As you can see, slicing works in a similar way for list, tuple, and range sequences.

Sequence Slicing

**5 Packing & Unpacking Sequences**

When you create a sequence, such as a list,tuple, range, or str, and assign values to it, this is called **packing.**

Conversely, when you extract values from a sequence into variables, it's called **unpacking.**

Here is an example:

**Packing and Unpacking a** tuple & **a** list

seasons= ('Spring', 'Summer', 'Fall', 'Winter') # *Packing a tuple*

season\_1, season\_2, season\_3, season\_4 = seasons # *Unpacking the tuple*

print(season\_2)

my\_grades = [95, 100]

stats\_grade, python\_grade = my\_grades

print(python\_grade)

# *Packing a list*

# *Unpacking the list*

**Sequence slicing** allows you to access a portion of a sequence (like a string, list, or tuple) by specifying a start, stop, and optional step. The syntax is [start:stop:step].

This yields:

Summer 100

If the number of variables on the left side of the assignment operator (=) doesn't match the number of elements in the sequence, Python will raise a ValueError.

Here are two examples:

**Unpacking too many values**

my\_grades = [95, 100, 80] stats\_grade, python\_grade my\_grades print(python\_grade)

# *Packing a list*

# *Unpacking the list*

yields

**ValueError:** too many values to unpack (expected 2)

and if you try

**Unpacking not enough values**

my\_grades = [95, 100, 80] # *Packing*

stats\_grade, python\_grade, om\_grade, dataviz\_grade my\_grades # *Unpacking*

print(python\_grade)

yields

**ValueError:** not enough values to unpack (expected 4, got 3)



One way to address this problem is using the unpacking operator \*· The\* operator allows you to unpack any iterable (list, tuple, string, etc.) into multiple variables.

Here's an example:

seasons= ('Spring', 'Summer', 'Fall', 'Winter') # *Packing a tuple* season\_1, \*middle\_seasons, last\_season = seasons # *Unpacking with\** print(middle\_seasons)

my\_grades = [95, 100, 80] # *Packing a list* stats\_grade, \*other\_grades my\_grades # *Unpacking with\** print(other\_grades)

This yields:

['Summer', 'Fall']

[100, 80]

In a similar fashion, you can also unpack strings and ranges. For example, you can extract individual characters from a string or values from a range.

Here is an example:

**Unpacking with\* in a** string **and a** range

str\_1 = 'Lunch time'

char\_1, \*chars, last char str 1 print(char\_1)

print(chars) print(last\_char)

print() # *an empty Line*

range\_1 = range(5)

\*r\_1, r\_2, r\_3 = range\_1 print(r\_1)

print(r\_2)

This yields:

L

**['u', 'n', 'c', 'h',** ' ' **'t', 'i', 'm']**

e

[0, 1, 2]

3

The variable with the \* operator can collect any number of values, including zero. If there aren't enough values to assign to variables without the \*, you'll get a ValueError.

Here's an example:

**Unpacking with insufficient number of values**

word= 'AI'

first, \*second, third, fourth word

print(f'first is {first}') print(f'second is {second}') print(f'third is {third}') print(f'fourth is {fourth}')

In this example, we're trying to unpack the word AI into four parts:

* the first variable will get A
* since the word only has two characters, the second will be an empty list [J
* **the** third **will get** I
* there are no remaining characters for the variable fourth, raising the following ValueError:

**ValueError:** not enough values to unpack (expected at least 3, got 2)

Packing & Unpacking Sequences

**Packing** refers to assigning multiple values to a single variable using a sequence like a list or tuple.

**Unpacking** is the reverse, where elements from a sequence are assigned to multiple variables.

1. **The zip** O & **enumerate O Functions**
   1. enumerate () **Function**

Looping through the elements of a sequence in Python is quite simple. Here's a basic example:

###### Iterating through a list

fruit= ['apple', 'banana', 'grape'] for item in fruit:

print(item)

This yields:

apple banana grape

However, sometimes you may need to iterate through both the indices and values of a sequence. There are two common ways to do this:

* + 1. **Using a for loop with** range O **(Less Pythonic)**

One way to access both indices and values is by using the range O function in combination with the len O function. This approach is more manual and prone to errors if not done carefully. Here's how it works:

###### Iterating with indices using range O

test\_1 = [20, 30, 40, 50]

**for** counter **in** range(len(test\_1)): print(counter, test\_1[counter])

This yields:

0 20

1 30

2 40

3 50

* + 1. **Using** enumerate O **function (More Pythonic)**

A cleaner and more Pythonic way to loop through both the indices and values of a sequence is by using Python's built-in enumerate O function. This function takes an iterable and gives you both the index and value for each item.

If you pass enumerate O into the list O function, you'll get a list of tuples, where each tuple contains an index and its corresponding value.

Here is an example:

**Using** enumerate() **with a** list()

test\_1 = [20, 30, 40, 50]

index\_value\_list = list(enumerate(test 1)) # *Creates a List of tuples*

print(index\_value\_list)

This returns:

[(0, 20), (1, 30), (2, 40), (3, 50)]

You can now loop through these index-value pairs using a for loop.

Here's an example:

**Iterating with** enumerate O

test\_1 = [20, 30, 40, 50]

**for** index, value **in** enumerate(test\_1): print(index, value)

In this snippet, the index, value pair is being unpacked from the tuple created by enumerate(). This means the index and value are separated into two variables directly inside the for loop. This yields:

0 20

1 30

2 40

3 50

You can also customize the starting index by using the optional start argument.

For example:

**Custom starting index with** enumerate O

This yields:

100 20

101 30

102 40

103 50

enumerate() Function

**6.2** zip O **Function**

The enumerate() function pairs each element of an iterable with its index. However, if you want to combine elements from two or more different iterables, you can use the zip() function.

The zip O function allows you to "bundle" multiple iterables together by pairing elements from each iterable at the same position, creating tuples for each corresponding set of elements. You can use the list O function to convert the output of the zip() function into a list.

Here's an example:

**Using** zip O **Function**

brides ['Emily', 'Olivia', 'Grace'] grooms ['William', 'Matthew', 'Anthony']

wedding\_pairs = list(zip(brides, grooms)) print(wedding\_pairs)

The **enumerate()** function adds a counter to an iterable, returning both the index and the value of each element in the iterable as you loop through it.

test\_1 = [20, 30, 40, 50]

**for** index, value **in** enumerate(test\_1, start=100): print(index, value)

This yields:

[('Emily', 'William'), ('Olivia', 'Matthew'), ('Grace', 'Anthony')]

In this example, the zip O function pairs each element from the brides list with the corresponding element from the grooms list, creating a *list of tuples.*

Here's how you can use zip O to pair more than two lists:

**Using** zip O **with Three Lists**

brides = ['Emily', 'Olivia' , 'Grace'] grooms= ['William', 'Matthew', 'Anthony'] wedding\_dates = ['Oct 30', 'Nov 7', 'Dec 4']

wedding\_details = list(zip(brides, grooms, wedding\_dates)) print(wedding\_details)

This yields:

[('Emily', 'William', 'Oct 30'), ('Olivia', 'Matthew', 'Nov 7'), ('Grace', 'Anthony', 'Dec 4')]

zip() Function

The **zip()** function takes multiple iterables and returns an iterator that aggregates elements from each iterable, pairing elements by their position. It stops when the shortest iterable is exhausted.

## Generator Expressions

#### What is a generator expression

Generator expressions are similar to list comprehensions in that they both provide a concise and elegant way to create sequences. According to python.org (https:/[/www.python.org/dev/peps/pep-0289/),generator](http://www.python.org/dev/peps/pep-0289/)%2Cgenerator) expressions are a high-performance, memory-efficient generalization of list comprehensions. The syntax for generator expressions is nearly identical to that of list comprehensions:

new\_generator\_exp = (expression for item in iterable if condition)

The key difference is that a generator expression is enclosed in parentheses C ) , whereas a list comprehension uses square brackets [ J.

Here's an example:

###### List Comprehension

list\_comprehension = [x \*\* 2 for x in range(4)] print(list\_comprehension)

This yields:

[0, 1, 4, 9]

**Generator Expression**

generator\_expression = (x \*\* 2 **for** x **in** range(4)) print(generator\_expression)

This yields:

<generator object <genexpr> at Ox000002719E3030B0>

As you can see, printing a generator expression doesn't immediately give us the values.

If you want to access the values, you can either:

1. **Iterate over the generator in a for loop**

**Iterating Over a Generator Expression**

generator\_expression = (x \*\* 2 **for** x **in** range(4))

**for** item **in** generator\_expression: print(item, end=' ')

This yields:

0 1 4 9

1. **Convert the generator to a list using the** list O **function**

**Convert Generator to List**

generator\_expression = (x \*\* 2 for x in range(4)) print(list(generator\_expression))

This yields:

[0, 1, 4, 9]

#### generator Expression Performance

The fundamental difference between list comprehensions and generator expressions is how they store the data. A list comprehension generates and stores the entire list in memory, whereas a generator expression generates the next element only when it is needed. For small sequences, the difference in memory usage is negligible. However, for large sequences, generator expressions can significantly improve memory efficiency.

**Memory Usage Comparison**

**import sys** # *module giving access to system specific parameters and functions*

list\_comp = [i **for** i **in** range(100\_000)] list\_comp\_size = sys.getsizeof(list\_comp)

gen\_exp = (i **for** i **in** range(100\_000)) gen\_exp\_size = sys.getsizeof(gen\_exp)

print(f'list\_comp\_size is {list\_comp\_size} bytes') print(f'gen\_exp\_size is {gen\_exp\_size} bytes')

This yields:

list\_comp\_size is 800984 bytes gen\_exp\_size is 112 bytes

As you can see, generator expressions are far more memory-efficient for large sequences.

We can also compare the speed of list comprehensions and generator expressions using the timeit module, which allows us to measure the execution time of small code snippets:

**Speed Comparison with** timeit

**import timeit**

list\_snippet =

I I I

list\_comp

I I I

[item for item in range(1\_000\_000) if item% 2

OJ

list time= timeit.timeit(list\_snippet, number=100) print(f'List comprehension time: {list\_time} seconds.')

generator\_snippet = '''

generator\_exp = (item for item in range(1\_000\_000) if item% 2 0)

I I I

generator\_time = timeit.timeit(generator\_snippet, number=100) print(f'Generator expression time: {generator\_time} seconds.')

This yields:

List comprehension time: 6.386 seconds. Generator expression time: 0.000039 seconds.

The generator expression is much faster because it creates values one at a time, instead of storing them all in memory at once.

#### List Comprehension vs Generator Expression: Which Should You Use?

For most cases, list comprehensions are more intuitive and easy to work with, especially when you need to access the entire list later. However, when you're dealing with large sequences or only need to calculate values once, generator expressions are more memory- and time-efficient.

Here's an example of using a generator expression to calculate the sum of squares of even numbers from Oto 9:

###### Using Generator with sum()

seq\_gen = (x \*\* 2 for x in range(10) if x % 2 0) result= sum(seq\_gen)

print(result)

This yields:

120

You could achieve the same result with list comprehensions, but using a generator expression is more efficient in this case.

To wrap up this section, remember that both list comprehensions and generator expressions are elegant and compact ways to define sequences. While list comprehensions store the entire sequence in memory, gen­ erator expressions generate elements one at a time needing less computational resources. For memory- or performance-sensitive applications, consider using generator expressions.

#### Tuple Comprehension: Is It Possible?

Generator expressions may bring to mind the idea of tuple comprehensions since they are enclosed in parenthe­ ses C ) . You might wonder: if we have list comprehensions, can we also create tuple comprehensions? Let's explore this by trying to create a tuple using a comprehension-like syntax.

###### Attempt 1: Without Parentheses

**Invalid Attempt: Tuple Comprehension without Parentheses**

# *Trying to create a tuple using List comprehension syntax*

attmpt\_1 = item **for** item **in** range(5) print(attmpt\_1)

This yields:

SyntaxError: invalid syntax

The attempt fails with a syntax error. Now, let's try again by adding parentheses.

**Attempt 2: With Parentheses (Generator Expression)**

**Attempt with Parentheses** C )

attempt\_2 = (item for item in range(5)) print(attempt\_2)

This yields:

<generator object <genexpr> at 0x0000025CDCC4F400>

Instead of a tuple, we get a generator object! As we learned earlier, parentheses are used for generator expressions, not for creating tuples. So **there is no tuple comprehension!**

The correct way to create a tuple using a comprehension-like syntax is by using Python's built-in tuple O function.

Here's how:

**Using** tuple O **Function**

tuple\_1 = tuple((item **for** item **in** range(5)))

# *ALternativeLy, you can omit the inner parentheses:* # *tupLe\_1* = *tupLe(item for item in range(S))* print(tuple\_1)

This yields:

(0, 1, 2, 3, 4)

We've now created a tuple using the tuple O function. Alternatively, we can also create a list using list comprehension and then convert it into a tuple per below example:

**Creating a Tuple from a List Comprehension**

tuple\_1 = tuple([item for item in range(5)]) print(tuple\_1)

This yields:

(0, 1, 2, 3, 4)

In this approach, we first use list comprehension to create a list, then use tuple O to convert it into a tuple.

Generator Expressions

**Generator expressions** are a compact way to create iterators. They work similarly to list comprehensions but generate items lazily, one at a time, using less memory. The syntax is the same as list comprehensions but with parentheses instead of brackets.

## Introducing Non-Sequence Data Types

So far, we have explored Python's collection data types that belong to the *sequence* category, including str, list, tuple, and range. These data types maintain the order of their elements, and each element is accessible using a zero-based index.

There are two common built-in *non-sequence* collection data types in Python:

* diet (dictionary)
* set

In non-sequence data types, the order of elements does not matter, and you cannot access them using indices.

This rest of this handout will cover the diet and set data types.

## Non-Sequence Data Type: diet

A dictionary10 stores data as **key-value pairs,** where each key maps to a value using a colon : . Keys must be unique and immutable (e.g., strings, numbers, or tuples), while values can be of any data type.

Here's an example:

**Creating a dictionary**

state\_capitals ={'VA': 'Richmond', 'NY': 'Albany', 'AZ': 'Phoenix'} print(state\_capitals)

This yields:

{'VA': 'Richmond', 'NY': 'Albany', 'AZ': 'Phoenix'}

In the above dictionary:

* Keys are 'VA'' 'NY'' and 'AZ'
* Values are 'Richmond', 'Albany', and 'Phoenix'

Each key is associated with a value using a colon : , and key-value pairs are separated by commas , . Unlike sequence types, dictionaries do not support indexing or slicing; instead, values are accessed directly through their corresponding keys.

In the following early example, we'll start with an empty dictionary {} and demonstrate how to add key-value pairs, check the number of elements using the len() function, check its type using type() function, and update the dictionary's values.

**Creating a** diet, **checking its type and length**

superhero\_dict = {} superhero\_dict['name'] = 'Thor' superhero\_dict['strength'] = 95

# *an empty dictionary*

# *Assign value of 'Thor' to 'name' key*

# *Assign value of 95 to 'strength' key*

superhero\_dict['abilities'] = ['lightning', 'hammer', 'flight']

10Dictionariesare also called *maps* or *associative arrays.*

print(type(superhero\_dict))

print(f'number of key-value pairs: {len(superhero\_dict)}') print(superhero\_dict)

This yields:

<class 'diet'>

number of key-value pairs: 3

{'name': 'Thor', 'strength': 95, 'abilities': ['lightning', 'hammer', 'flight']}

A dictionary is **mutable,** meaning you can modify it after creation. You can add new key-value pairs, update existing ones, or remove them without needing to create a new dictionary.

Here is an example:

**Modifying a** diet

superhero\_dict['strength'] = 98 # *Update existing key-value pair* superhero\_dict['intelligence'] 85 *#Adda new key-value pair* print(superhero\_dict)

This yields:

{'name': 'Thor', 'strength': 98, 'abilities': ['lightning', 'hammer', 'flight'], 'intelligence': 85}

Besides len O and type O, other built-in functions like sorted O can also be used with dictionaries. By default, these functions operate on the dictionary's keys11.

In the example below, we apply three common built-in functions to a dictionary:

###### Using Built-in Functions

superhero\_strengths ={'Spider-Man': 75, 'Iron Man': 85, 'Thor': 98}

print(len(superhero\_strengths))

print(sorted(superhero\_strengths))

# *number of key-value pairs 3*

# *Outputs sorted keys*

print(sum(superhero\_strengths.values())) # *Outputs sum of values: 258*

# *next section will cover values() method*

This yields:

3

['Iron Man', 'Spider-Man', 'Thor'] 258

11Unless explicitly specified to apply to the values.

Dictionary diet

A **dictionary** {diet) is a collection of key-value pairs. Each key is unique, and it allows fast access to values by using the key. Dictionaries are unordered and mutable.

#### Dictionary Operations

Since the dictionary class is mutable, it offers a variety of methods for accessing and updating its contents. This allows you to modify, add, or remove key-value pairs as needed. The table below summarizes the most important dictionary methods12 and operations:

|  |  |  |
| --- | --- | --- |
| **Method** | **Sample Syntax** | **Description** |
| get() | dict.get(key, default) | Returns value for key, or default if not found. |
| keys() | diet. keys() | Returns a view object of all the keys in the dictionary. |
| values() | diet. values() | Returns a view object of all the values in the dictionary. |
| i terns() | diet. i terns() | Returns a view object of key-value pairs as tuples. |
| copy() | diet. copy() | Returns a shallow copy of the dictionary. |
| update() | dict.update(other\_dict) | Updates dictionary with key-value pairs from another dictionary  or iterable. |
| clear() | diet. clear() | Removes all elements from the dictionary. |

Table 4: Important Dictionary Methods

* 1. get O **Method for Accessing** diet **Values**

You can access a value by placing the key inside square brackets [J, like Dog\_dict ['age' J. However, if the key does not exist in the dictionary, this will raise a KeyError. To avoid this risk, you can use the get O method, which safely returns None or a custom message if the key is not found, ensuring no errors occur.

**Using** get O **Method**

Dog\_dict {'name': 'Milo',

'age': 9,

'surgeries': ['jaw', '1-leg', 'r-leg']

}

print(Dog\_dict.get('age')) # *Returns 9*

print(Dog\_dict.get('Vet', 'Key not found')) # *Returns 'Key not found'*

12There are less used methods such as fromkeysO, pop, popitem, and setdefault and you are encouraged to explore them yourself.

|  |  |  |
| --- | --- | --- |
| **Operator/Keyword** | **Sample Syntax** | **Description** |
| in | key in diet | Returns True if the key is in the dictionary. |
| not in | key not in diet | Returns True if the key is not in the dictionary. |
| -- | diet1 ==diet2 | Returns True if two dictionaries have the same  key-value pairs. |
| != | diet1 != diet2 | Returns True if two dictionaries are not equal. |
| del | del diet [key]  del diet | Deletes the specified key-value pair from the  dictionary. Can also delete the entire dictionary. |
| I | diet1 I diet2 | Returns a new dictionary by merging two  dictionaries. |
| \*\* | diet 1 = { \*\*diet2, \*\*diet3} | Merges two dictionaries using unpacking. |
| \* | \*diet | Unpacks keys of the dictionary.  Used inside a function such as print C\*diet) |

Table 5: Important Dictionary Operators and Keywords

This yields:

9

Key not found

* 1. keys O **Method**

The keys() method returns a view object13 that displays all the keys in the dictionary. This allows you to loop through the keys directly, as shown in the example below:

# t=+il:\-19¥9:':Mi®i►

or kin Dog\_diet.keys():

print(k)

This yields:

name age

surgeries

You cannot modify the keys of a dictionary by calling the keys O method because it only provides a view object

13A view object shows the current keys in the dictionary. If the dictionary is changed, the view object will automatically update to show the changes.

of the current keys. If you attempt to directly change the keys via this view object, you will get an error:

**Attempt to modify keys by accessing them using** keys O **method**

Dog\_dict ={'name': 'Milo', 'age': 9} Dog\_keys = Dog\_dict.keys()

Dog\_keys[O] = 'breed' # *Attempting to modify the first key*

This yields:

TypeError: 'dict\_keys' object does not support item assignment

However, if you want to get a list of all the keys, you can convert the view object returned by keys O into a list using the list O function

**Converting keys to a list**

Dog\_dict ={'name': 'Milo', 'age': 9}

keys\_list = list(Dog\_dict.keys()) # *Convert keys to a List*

print(keys\_list)

This yields:

['name', 'age']

* 1. values() **Method**

The values O method returns a view object displaying all the values in the dictionary. It allows you to loop through the values directly, as shown in the example below. Similar to the keys O method, the values cannot be modified through the values O method because it only provides a view of the current values.

**Using** values() **Method**

Dog\_dict {'name': 'Milo',

'age': 9,

'surgeries': ['jaw', '1-leg', 'r-leg']

}

**for** v **in** Dog\_dict.values(): print(v)

This yields:

'Milo'

9

['jaw', '1-leg', 'r-leg']

Just like with the keys O method, you can use the list O function to get a list of all the values in the dictionary:

**Converting values to a** list

values\_list = list(Dog\_dict.values()) # *Convert values to a List*

print(values\_list)

This yields:

['Milo', 9, ['jaw', '1-leg', 'r-leg']]

* 1. i terns O **Method**

The iterns O method returns a view object containing the key-value pairs of the dictionary as *tuples.* This makes it easy to loop through both the keys and their associated values simultaneously, which can be especially useful when you need to work with both parts of the dictionary in one loop.

**Using** iterns O **Method**

fork, v in Dog\_dict.iterns(): print(f'Key = {k}, Value= {v}')

This yields:

Key name, Value= Milo Key age, Value= 9

Key surgeries, Value= ['jaw', '1-leg', 'r-leg']

In this example, the iterns O method returns a sequence of tuples, where each tuple contains a key and its corresponding value from the dictionary. The loop variables k and v represent the key and value, re­ spectively. This process of assigning key-value pairs from tuples into separate variables is known as **tuple unpacking** (that we have already covered in the previous section). By looping over the result of iterns O, you can efficiently handle both the dictionary's keys and values.

You can also convert the key-value pairs into a list of tuples using the list O function:

**Converting items to a list of tuples**

iterns\_list = list(Dog\_dict.iterns()) # *Convert items to a List of tuples*

print(iterns\_list)

This yields:

[('name', 'Milo'), ('age', 9), ('surgeries', ['jaw', '1-leg', 'r-leg'])]

* 1. copy O **Method**

The copy O method creates a shallow copy of the dictionary.14

**Using** copy O **Method**

Dog\_dict ={'name': 'Milo', 'age': 9, 'surgeries': ['jaw', '1-leg', 'r-leg']} copy\_dict = Dog\_dict.copy() # *Create a shallow copy*

print(copy\_dict)

This yields:

'name': 'Milo', 'age': 9, 'surgeries': ['jaw', '1-leg', 'r-leg']

* 1. update() **Method**

The update O method allows you to merge two dictionaries by adding key-value pairs from one dictionary into another. If a key already exists in the original dictionary, its value will be updated with the new one from the second dictionary. This is an **in-place**15 modification of the dictionary.

**Using** update O **method**

program\_dict\_1 = {'BARM': 50, 'Marketing': 150}

program\_dict\_2 ={'Finance': 200, 'HCM': 250, 'BARM': 1000} program\_dict\_1.update(program\_dict\_2)

print(program\_dict\_1)

In the example below, program\_dict\_1 is updated with the key-value pairs from program\_dict\_2, including the 'BARM' key, which is overwritten. This yields:

{'BARM': 1000, 'Marketing': 150, 'Finance': 200, 'HCM': 250}

The update O method can also accept an iterable of key-value pairs, such as a list of tuples, making it possible to update a dictionary from non-dictionary sources.

**Using** update O **with an lterable**

dict\_E = {'BARM': 100, 'Marketing': 150}

my\_list = [('Finance', 120), ('HCM', 40)] # *List of tuples*

dict\_E.update(my\_list) print(dict\_E)

14This means the new dictionary is a separate object, but the values inside it (if they are mutable objects like lists) are still references to the same objects in memory as the original dictionary. So, if you modify a mutable object (like a list, or another dictionary) in one dictionary, those changes will appear in both the original and the copied dictionary. For example, adding an item to a list value in the copied dictionary will also add that item to the list in the original dictionary, because they both point to the same list in memory. We will cover this in more detail later.

15You modify the original dictionary without creating a new one

In this example, dict\_E is updated with key-value pairs from my \_list: This yields:

{'BARM': 100, 'Marketing': 150, 'Finance': 120, 'HCM': 40}

This is a good opportunity to introduce two other popular methods for merging dictionaries.

#### Method 1: Using I (Union Operator)

Starting in Python 3.9, the union operator I can be used to merge two or more dictionaries:

###### Merging Dictionaries with I

diet A {'Python': 100, 'Simulation': 99} diet B {'BA' : 95, 'Python' : 90}

diet C {'Statistics' : 85, 'OM' : 70, 'Python' : 9000}

dict\_D diet A diet B diet C print(dict\_D)

This yields:

{'Python': 9000, 'Simulation': 99, 'BA': 95, 'Statistics': 85, 'OM': 70}

You can merge more than two dictionaries using I. If a key is repeated in multiple dictionaries, the value from the last dictionary will be retained. In this example, the value of 'Python' from dict\_C overrides the others.

#### Method 2: Using\*\* (Unpacking Operator)

Another way to merge dictionaries is by using the unpacking operator\*\*:

###### Merging Dictionaries with \*\*

diet A {'Python': 100, 'Simulation': 99} diet B {'BA' : 95, 'Python' : 90}

diet C {'Statistics' : 85, 'OM' : 70, 'Python' : 9000}

merged\_dict = {\*\*dict\_A, \*\*dict\_B, \*\*dict\_C} print(merged\_dict)

This yields:

{'Python': 9000, 'Simulation': 99, 'BA': 95, 'Statistics': 85, 'OM': 70}

* 1. clear O **Method**

The clear O method removes all key-value pairs from the dictionary, leaving it empty. After calling this method, the dictionary is still available but contains no elements, as shown in the example below.

clear O **method**

test\_dict = {'k1': 10, 'k2': 20} test\_dict.clear() print(test\_dict)

This yields:

{}

* 1. **Using in and** not in **operators**

The in and not in operators check for the presence or absence of a **key** in a dictionary. The in operator returns

True if the specified key is found, while not in returns True if the key is not found.

**Using in and** not in **operators**

Dog\_dict ={'name': 'Milo', 'age': 9, 'surgeries', ['jaw', '1-leg', 'r-leg']}

print('name' in Dog\_dict) print('Vet' in Dog\_dict) print('Vet' not in Dog\_dict)

# *Check if 'name' is a key*

# *Check if 'Vet' is a key*

# *Check if 'Vet' is not a key*

This yields:

True False True

#### Using== and != operators

The == operator checks if two dictionaries contain the same key-value pairs, regardless of the order. The ! =

operator checks if the dictionaries are not equal, returning True if there is any difference between them.

**Comparing dictionaries**

diet A {'name' : 'Milo' , 'age' : 9}

diet B {'age': 9, 'name': 'Milo'}

diet C {'name': 'Milo', 'age': 9, 'surgeries': ['jaw', '1-leg', 'r-leg']} print(dict\_A diet B) # *Should be True*

I print(dict\_B != dict\_C) # *Should be True*

This yields:

True True

* 1. **Using** del **Keyword**

The del keyword can be used to remove a specific key-value pair from a dictionary or delete the entire dictionary from memory. When a key is deleted, both the key and its associated value are removed. If you delete the entire dictionary, it no longer exists, and any attempt to access it will raise a NameError.

**Using** del **to remove a key-value pair**

Dog\_dict ={'name': 'Milo', 'age': 9, 'surgeries': ['jaw', '1-leg', 'r-leg']}

del Dog\_dict['age'] # *Removes the key 'age' and its value*

print(Dog\_dict)

This yields:

{'name': 'Milo', 'surgeries': ['jaw', '1-leg', 'r-leg']}

You can also delete the entire dictionary using the del keyword, as shown in the following example:

**Deleting the entire dictionary using** del **keyword**

Dog\_dict ={'name': 'Milo', 'age': 9, 'surgeries': ['jaw', '1-leg', 'r-leg']}

del Dog\_dict

print(Dog\_dict)

# *Completely removes Dog\_dict*

This yields:

NameError: name 'Dog\_dict' is not defined

#### Using\* Operator to Display Dictionary Keys

Using the \* operator on a dictionary will display its keys.

Here's an example:

**Using\* to Display Dictionary Keys**

dog= {'name': 'Milo', 'age': 9, 'surgeries': ['jaw']} print(\*dog) # *displaying dictionary keys*

This yields:

name age surgeries

* 1. diet() **function and creating a dictionary with** zip() **function**

So far, we have seen how to create a dictionary by packing it directly with key-value pairs. Python also provides the diet O function, which is another way to create a dictionary. This function works similarly to other type­ conversion functions like str(), int(), float(), list(), and tuple(). You can use diet O to build dictionaries from key-value pairs or iterable objects such as lists of tuples, as shown in the below example:

**Using** diet O **function**

dict\_1 = dict(name

print(dict\_1)

"James", age

25) # *Creating a diet with key-value pairs*

list\_of\_tuples = [('x', 1), ('y', 2), ('z', 3)]

dict\_2 = dict(list\_of\_tuples) # *Creating a diet with a list of tuples*

print(dict\_2)

This yields:

{'name': 'John', 'age': 30, 'job': 'Analyst'}

{'x': 1, 'y': 2, 'z': 3}

Another useful method for creating dictionaries is by using the zip O function. This function combines two lists or other iterables into pairs, and then diet O can turn these pairs into key-value pairs for the dictionary. Here's an example:

**Creating a dictionary with** zip O

keys = ['name', 'age' , 'job']

values= ['Milo', 9, 'Chief Barking Officer'] new\_dict = dict(zip(keys, values)) print(new\_dict)

This yields:

{'name': 'Milo', 'age': 9, 'job': 'Chief Barking Officer'}

The zip() function combines the elements from the keys and values lists to create key-value pairs, which are

then used to build the new dictionary using the diet O function.

#### Dictionary Comprehension

So far, we have seen several ways of creating dictionaries, such as manually assigning key-value pairs, using the update O method, leveraging the zip O function, and utilizing the diet O function. However, **dictionary com­ prehension** provides yet another powerful and concise way to create dictionaries by transforming and filtering existing data.

Similar to list comprehension, dictionary comprehension allows you to generate dictionaries by iterating over existing ones, optionally applying conditions or transformations to keys and values. The general template for dictionary comprehension is:

new\_dict ={key: value for (key, value) in old\_dict. i terns() if condition}

This structure allows you to conditionally include items from old\_dict in the new dictionary, while also transform­ ing keys and values as needed. Let's explore a few examples.

#### Modifying the Values

In this example, we have a dictionary of action movie heroes with their power stats. We create a new dictionary by doubling the power stat for each hero:



hero stats = {"John Wick": 85, "Ethan Hunt": 90, "Lara Croft": 75, "Jason Bourne": 80} boosted\_stats = {k: v \* 2 fork, v in hero\_stats.items()}

print(boosted\_stats)

This yields:

{'John Wick': 170, 'Ethan Hunt': 180, 'Lara Croft': 150, 'Jason Bourne': 160}

#### Modifying the Keys

Here, we modify the names of the heroes by adding a title "Agent" to each name:

**Modifying keys**

hero\_stats = {"John Wick": 85, "Ethan Hunt": 90, "Lara Croft": 75, "Jason Bourne": 80} agent\_heroes = {"Agent " + k: v for k, v in hero\_stats.items()}

print(agent\_heroes)

This yields:

{'Agent John Wick': 85, 'Agent Ethan Hunt': 90, 'Agent Lara Croft': 75, 'Agent Jason Bourne': 80}

#### Modifying Both Keys and Values

We can also create a new dictionary where both the keys and values are modified:

**Modifying Both keys and values**

hero\_stats = {"John Wick": 85, "Ethan Hunt": 90, "Lara Croft": 75, "Jason Bourne": 80}

super\_agent\_stats = {"Agent " + k: v \* 2 fork, v in hero\_stats.items()} print(super\_agent\_stats)

This yields:

'Agent John Wick': 170, 'Agent Ethan Hunt': 180, 'Agent Lara Croft': 150, 'Agent Jason Bourne': 160

#### Adding Conditions

We can also add filters using conditionals. Here is an example:

**Conditional Comprehension**

hero\_stats = {"John Wick": 85, "Ethan Hunt": 90, "Lara Croft": 75, "Jason Bourne": 80} elite\_heroes = {k: v **fork,** v **in** hero\_stats.items() **if** v > 80}

print(elite\_heroes)

This yields:

'John Wick': 85, 'Ethan Hunt': 90

#### Handling Missing Keys or Values with None Keyword

Missing data is a common part of working with real-world datasets, and dictionaries can effectively handle such cases by assigning None to indicate missing values or even using it as a key. None is a special reserved keyword that represents the absence of a value.

**Using** None **in a Dictionary**

dict\_A = {'Monday' : **None,** 'Tuesday' : 0, 'Wednesday' : **None,** 'Thursday' : 3} print(dict\_A)

This yields:

{'Monday': None, 'Tuesday': 0, 'Wednesday': None, 'Thursday': 3}

None is not equal to o; it means "no value." You can use is None or is not None to check for None values. Avoid using == for this check.16 Later, we will encounter the null value for missing data in Pandas.

16There is a difference between == and is. The former checks for *value equality,* while the latter checks for *reference equality.* Read

Here's an example of using dictionary comprehension to remove items with None values:

**Removing** None **Values**

dict\_A = {'Monday': None, 'Tuesday': 0, 'Wednesday' : None, 'Thursday': 3} dict\_B = {k: v fork, v in dict\_A.items() if vis not None} print(dict\_B)

This yields:

{'Tuesday': 0, 'Thursday': 3}

#### Copying Dictionaries

You cannot copy a dictionary using dictB = dictA. This operation will create a reference from dictB to dictA,

meaning that any changes made to dictA will also affect dictB.

The example below demonstrates this behavior:

**Dictionary Assignment Reference**

dict\_A ={'Math': 95, 'Science': 100} dict\_B = dict\_A # *Reference, not a copy* dict\_A.clear() # *Clear dict\_A* print(f'dict\_A is {diet A}') print(f'dict\_B is {diet B}')

This yields:

diet A is{} diet Bis{}

Even though you didn't change dictB directly, since dictA and dictB refer to the same object, modifying one also affects the other.

#### Two Ways to Copy a Dictionary

There are two ways to make a copy of a dictionary:

* Using the diet O function
* Using the copy O method

Here's how you can create a *shallow* copy using these methods:

morealhttps://stackoverflow.com/questions/132988/is-there-a-difference-between-and-is

**Copying a Dictionary**

dict\_A ={'Python': 95, 'Statistics': 100}

dict\_B = dict\_A.copy() # *Alternatively, use dict\_B* = *dict(dict\_A)*

dict\_A.clear() # *Clear dict\_A* print(f'dict\_A is {diet A}') print(f'dict\_B is {diet B}')

This yields:

diet A is{}

diet Bis {'Python': 95, 'Statistics': 100}

As shown, modifying dictA does not affect dictB because copy O creates a new, independent dictionary.

#### Nested Dictionaries

A nested dictionary is a dictionary that contains other dictionaries as its values. This structure allows you to or­ ganize complex data hierarchically. The following example demonstrates how to create a dictionary that contains two other dictionaries as values.

**Creating a Nested Dictionary**

pet\_1 ={'name': 'Milo', 'age': 9}

pet\_2 ={'name': 'Pooh', 'age': 100} my\_pets ={'dog': pet\_1, 'bear': pet\_2}

print(my\_pets)

This yields:

{'dog': {'name': 'Milo', 'age': 9}, 'bear': {'name': 'Pooh', 'age': 100}}

In this example, the outer dictionary my\_pets has two keys, 'dog' and 'bear', each containing its own dictionary as a value.

### List of Dictionaries for Tabular Data

A common way to represent tabular data in Python is using a list of dictionaries. In this structure, each dictionary represents a row of the table, and the dictionary keys serve as the column headers. This makes it easy to work with datasets where each row contains multiple attributes.

Here's an example where we represent a simple table of student scores using a list of dictionaries:

This table can be represented as a list of dictionaries, with each dictionary storing the information for one student:

|  |  |  |
| --- | --- | --- |
| **Name** | **Python** | **Statistics** |
| Dwayne Johnson | 95 | 90 |
| Scarlett Johansson | 88 | 85 |
| Robert Downey | 91 | 93 |
| Emma Watson | 87 | 89 |

Table 6: Celebrity Python and Statistics Scores

###### List of Dictionaries

students = [

{'Name': 'Dwayne Johnson', 'Python': 95, 'Statistics': 90},

{'Name': 'Scarlett Johansson', 'Python': 88, 'Statistics': 85},

{'Name': 'Robert Downey', 'Python': 91, 'Statistics': 93},

{'Name': 'Emma Watson', 'Python': 87, 'Statistics': 89}

print(students)

This yields:

['Name': 'Dwayne Johnson', 'Python': 95, 'Statistics': 90, 'Name': 'Scarlett Johansson', 'Python': 88, 'Statistics': 85, 'Name': 'Robert Downey', 'Python': 91, 'Statistics': 93, 'Name': 'Emma Watson', 'Python': 87, 'Statistics': 89]

This format is easy to use for a variety of data processing tasks, such as filtering rows, calculating averages, or extracting information. For example, to get the average Python score of all students:

###### Calculating Average Python Score

average\_python = sum(student['Python'] **for** student **in** students) / len(students) print(f'Average Python score: {average\_python}')

This yields:

Average Python score: 90.25

You can also extract specific columns, such as all student names:

###### Extracting a Column from List of Dictionaries

names= [student['Name'] for student in students] print(names)

This yields:

['Dwayne Johnson', 'Scarlett Johansson', 'Robert Downey', 'Emma Watson']

This structure is especially helpful when working with datasets that are easily represented as rows and columns, such as CSV or JSON files, and can be easily converted to other formats, such as Pandas DataFrames.

## Shallow vs. Deep Copy

Throughout this handout, we've mentioned shallow and deep copies. In this section, we'll explore the difference between them. The copy() method creates a *shallow copy* of a dictionary. This means that the dictionary itself is copied, but any objects referenced by its values (like lists or dictionaries) are still shared between the original and the copy. As a result, changes made to these shared objects will affect both the original and the copied dictionary.

Here's an example:



original = {'a': 1, 'b': [2, 3, 4]} shallow\_copy = original.copy()

shallow\_copy['b'][OJ = 100 # *change from 2 to 100* print(f'original is: {original}') print(f'shallow\_copy is: {shallow\_copy}')

yields

original is: {'a': 1, 'b': [100, 3, 4]}

shallow\_copy is: {'a': 1, 'b': [100, 3, 4]}

As you can see, modifying the list in the shallow copy also affects the original dictionary because both are pointing to the same list. This may cause issues because changes in one dictionary could unintentionally affect the other, leading to unexpected behavior or bugs in your program when you assume they are independent.

To create a *deep copy,* where all nested objects are fully copied (not just referenced), you can use the deep copy() function from the copy module. With a deep copy, changes to the copied object will not affect the original.

Here's an example:

**Using** deepcopy()

import copy

original = {'a': 1, 'b': [2, 3, 4]} deep\_copy = copy.deepcopy(original)

print(f'original is:

{original}')

print(f'shallow\_copy is: {shallow\_copy}')

yields

original is: {'a': 1, 'b': [2, 3, 4]}

shallow\_copy is: {'a': 1, 'b': [100, 3, 4]}

As shown above, modifying the list in the deep copy does not affect the original dictionary, because all objects, including the nested list, have been fully copied.

For more information about copy O and deepcopy O, you can refer to the official Python documentation:

https://docs.python.org/3/library/copy.html. diet() Function



**12 Non-Sequence Data Types: set**

Sets are used to store unordered, un-indexed *unique* values. A set is created by enclosing its elements inside curly brackets, {}.

games\_A {'Mario', 'Zelda', 'Sonic'}# *no repetition*

games\_B {'FIFA', 'Halo', 'FIFA', 'Minecraft', 'Mario', 'Halo'}# *with repetition*

print(games\_A) print(games\_B)

The **diet()** function in Python is used to create a new dictionary. It can either create an empty dictionary or convert a sequence of key-value pairs into a dictionary.

This yields:

{'Mario', 'Sonic', 'Zelda'}

{'Mario', 'FIFA', 'Minecraft', 'Halo'}

As you may have noticed, duplicate values were ignored without causing an error. This property of a set can be used for duplicate elimination.

Similar to dictionaries, sets are not sequences; hence, they do not support indexing and slicing using square brackets, [J. A set is **mutable;** you can add and remove elements from a set, but set elements themselves must be immutable (e.g., str, int, float, and tuple containing only immutable elements). Once an element is assigned to a set, you cannot change that specific element.

* 1. set **Methods**

The table below lists common methods for a set, followed by a few examples.

|  |  |  |
| --- | --- | --- |
| **Method** | **Sample Syntax** | **Description** |
| set() | my\_set = set() | Create an empty set |
| add() | set.add(item) | Add itern to the set |
| update() | set.update(iterable) | Add multiple elements from iterable |
| remove() | set.remove(item) | Remove itern from the set. Raises an error if itern is not found |
| discard() | set.discard(item) | Remove itern if it exists. Does not raise an error if itern is not found |
| clear() | set.clear() | Remove all elements from the set |

Table 7: Common methods for set

**Creating an Empty Set**

my\_empty\_set = set() # *empty set*

print(my\_empty\_set)

This yields:

set()

**Adding an Element**

my\_set ={'Python', 'Java'} my\_set.add('R') print(my\_set)

This yields:

{'Python', 'R', 'Java'}

**Adding Multiple Elements**

my\_set ={'Python', 'Java'} my\_set.update(['R', 'C++']) print(my\_set)

This yields:

{'Python', 'R', 'C++', 'Java'}

**Checking Length of a Set**

my\_set ={'Python', 'R', 'Java'} print(len(my\_set))

This yields:

3

**Removing an Element**

my\_set ={'Python', 'R', 'Java'} my\_set.remove('R') print(my\_set)

This yields:

{'Python', 'Java'}

**Discarding an Element**

my\_set ={'Python', 'R', 'Java'}

my\_set.discard('C++') # *No error even though 'C++' is not in the set*

print(my\_set)

This yields:

{'Python', 'R', 'Java'}

**Clearing All Elements**

my\_set ={'Python', 'R', 'Java'} my\_set.clear()

print(my\_set)

This yields:

set()

#### Set Operators and Keywords

Besides the in, not in, ==, and ! = operators, as well as the del keyword that we are already familiar with, Python's set data type has specific operators. These operators are listed in the table below.

Let's see a few examples

|  |  |  |
| --- | --- | --- |
| **Operator** | **Sample Syntax** | **Description** |
| I | set\_A I set\_B | **Union:** Returns a set containing all elements from both sets. |
| & | set\_A & set\_B | **Intersection:** Returns elements present in both sets. |
| - | set- A - set- B | **Difference:** Returns elements in set\_A and not in set- B. |
| - | set- A -set- B | **Symmetric difference:** Returns elements in either set, but not both. |
| <= | set\_A <= set\_B | **Subset:** Returns True if set\_A is a subset of set\_B. |
| >= | set\_A >= set\_B | **Superset:** Returns True if set\_A is a superset of set\_B. |
| < | set\_A < set\_B | **Strict subset:** Returns True if set\_A is a proper subset of set\_B. |
| > | set\_A > set\_B | **Strict superset:** Returns True if set\_A is a proper superset of set\_B. |

Table 8: Set operators and keywords in Python

###### Union I Operator

superheroes= {'Iron Man', 'Captain America', 'Black Panther'} video\_games = {'Call of Duty', 'Fortnite', 'Black Panther'} union\_set = superheroes I video\_games

print(f'Union of sets: {union\_set}')

This yields:

Union of sets: {'Iron Man', 'Fortnite', 'Captain America', 'Black Panther', 'Call of Duty'}

###### Intersection & Operator

superheroes= {'Iron Man', 'Captain America', 'Black Panther'} video\_games = {'Call of Duty', 'Black Panther', 'Fortnite'} intersection\_set = superheroes & video\_games print(f'Intersection of sets: {intersection\_set}')

This yields:

Intersection of sets: {'Black Panther'}

###### Difference - Operator

superheroes= {'Iron Man', 'Captain America', 'Black Panther'} video\_games = {'Call of Duty', 'Fortnite', 'Black Panther'} difference\_set = superheroes - video\_games

print(f'Difference of sets: {difference\_set}')

**This yields:**

Difference of sets: {'Iron Man', 'Captain America'}

**Symmetric Difference- Operator**

superheroes= {'Iron Man', 'Captain America', 'Black Panther'} video\_games = {'Call of Duty', 'Fortnite', 'Black Panther'} symmetric\_difference\_set = superheroes - video\_games print(f'Symmetric Difference of sets: {symmetric\_difference\_set}')

###### This yields:

Symmetric Difference of sets: {'Iron Man', 'Fortnite', 'Captain America', 'Call of Duty'}

**Subset <= Operator**

avengers= {'Iron Man', 'Captain America'}

all\_heroes = {'Iron Man', 'Captain America', 'Black Panther', 'Thor'} print(f'Is avengers a subset of all\_heroes? {avengers<= all\_heroes}')

###### This yields:

Is avengers a subset of all\_heroes? True

**Strict Subset < Operator**

avengers= {'Iron Man', 'Captain America'}

all\_heroes = {'Iron Man', 'Captain America', 'Black Panther', 'Thor'} print(f'Is avengers a strict subset of all\_heroes? {avengers< all\_heroes}')

###### This yields:

Is avengers a strict subset of all\_heroes? True

**Superset >= Operator**

avengers= {'Iron Man', 'Captain America'}

all\_heroes = {'Iron Man', 'Captain America', 'Black Panther', 'Thor'} print(f'Is all\_heroes a superset of avengers? {all\_heroes >= avengers}')

###### This yields:

Is all\_heroes a superset of avengers? True

#### Creating a Set from Other Collections

You can create a set from other collections such as lists, tuples, dictionaries, or ranges using the built-in set O function:

###### Creating Set from Other Collections

# *From a List of favorite colors*

favorite\_colors = ['Purple', 'Orange', 'Green', 'Purple', 'Blue'] color\_set = set(favorite\_colors)

print(f'color\_set = {color\_set}')

# *From a tuple of movie genres*

movie\_genres = ('Action', 'Romantic Comedy') genre\_set = set(movie\_genres) print(f'genre\_set = {genre\_set}')

# *From dictionary values (favorite streaming platforms)*

streaming\_services ={'Alex': 'Netflix', 'Sara': 'Disney+', 'Emma': 'Netflix'} streaming\_set = set(streaming\_services.values())

print(f'streaming\_set = {streaming\_set}')

# *From a range of episode numbers* episode\_range = range(1, 6) # *Episodes 1 to 5* episode\_set = set(episode\_range) print(f'episode\_set = {episode\_set}')

This yields:

color\_set ={'Purple', 'Orange', 'Green', 'Blue'} genre\_set ={'Action', 'Romantic Comedy'} streaming\_set = {'Netflix', 'Disney+'} episode\_set = {1, 2, 3, 4, 5}



**A set** is an unordered collection of unique elements. Sets do not allow duplicates and support opera­ tions like union, intersection, and difference. They are mutable and useful for removing duplicates and performing membership tests efficiently.

### Non-Sequence Data Types: frozenset as an Immutable Version of set

frozenset O creates an **immutable** set. Once a frozenset is created, it cannot be modified, but it can still be used for mathematical operations.

**Using** frozenset

IDs = {' 1234', '6789' , '5678'}

fz\_IDs = frozenset(IDs) print(fz\_IDs)

fz IDs.add('0099') # *Attempting to add will raise an error*

This yields:

frozenset({'6789', '1234', '5678'})

followed up

**AttributeError:** 'frozenset' object has no attribute 'add'

A **frozenset** is an immutable version of a set. Like sets, it contains unique elements and supports set operations like union, intersection, and difference. However, once created, elements cannot be added or removed from a frozenset.

frozenset

The following table provides a concise comparison of various Python collection data types we have discussed in this handout. It serves as a quick reference to understand how each type behaves in terms of iteration, duplication, and structure within Python.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Type** | **lterable** | **Mutable** | **Sequence** | **Indexable** | **Ordered** | **Supports**  **Slicing** | **Allows**  **Duplicates** |
| **String**  str | Yes | No | Yes | Yes | Yes | Yes | Yes |
| **Range**  range | Yes | No | Yes | Yes | Yes | Yes | No |
| **List**  list | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| **Tuple**  tuple | Yes | No | Yes | Yes | Yes | Yes | Yes |
| **Dictionary**  diet | Yes | Yes | No | No | **No\*** | No | Keys: No  Values: Yes |
| **Set**  set | Yes | Yes | No | No | **No\*** | No | No |
| **Frozen Set**  frozenset | Yes | No | No | No | No | No | No |

\*Note: Starting from Python 3.7, diet and set preserve the insertion order.

Table 9: Comparison of Python Data Types

Next Topic Preview

In the next handout, we will focus on creating your own custom functions, enabling you to define reusable blocks of code to solve specific problems. Additionally, we will introduce the powerful pandas library, which is essential for efficiently handling and analyzing structured data in Python. Topics will include the need for pandas, as well as data manipulation, filtering, and aggregation using real-world datasets.

### Exercises

* 1. What is the output of running this code?

###### What is the output?

list\_1 = []

for item inrange(3): list\_1[item] .append(item)

print(item) print (list\_1)

* 1. Create a one-dimensional array (a list or a tuple) with 10 integer numbers and try to access the item at index 10. What happens when you run your code?
  2. Create a tuple with 5 items. Convert the tuple into a list, add two more items, and convert it back to a tuple. What are the final contents of the tuple?
  3. Create a tuple with 5 numbers, then try to change the third number in the tuple to another value. What happens? How can you "modify" the tuple while keeping its immutability intact?
  4. Consider the following snippet

# fril3i:hMi:Sf♦

tr\_a = "Python is fun!"

Use slicing to reverse the str, then extract only the characters from the 4th to the 8th position (inclusive) in the reversed str. What is the result?

* 1. Create a range O that starts from 5 and ends at 25, incrementing by 3. Convert this range into a list. Can you change the 5th item in the list to 100? What happens if you try to change the item directly in the range?
  2. Use the range O function to generate a sequence of numbers from O to 100 (inclusive) with a step of 10. Convert this sequence into a list and calculate the sum of the elements. What is the result?
  3. You are given a list of daily sales figures for a week:



sales= [100, 150, 200, 250, 300, 350, 400]

Write a snippet that calculates the cumulative sum (cumsum) of the sales list using basic list operations (i.e., without using any external libraries like numpy or Python's built-in i tertools functions). **Hint:** rely on append() method inside a for loop. The output list should be:



cumsum sales= [100, 250, 450, 700, 1000, 1350, 1750]

**As an additional challenge:** write a snippet that calculates the running average of the sales figures as well. The running average up to each day is the cumulative sum divided by the number of days up to that

point. The ideal output would be:



running\_avg\_sales [100, 125, 150, 175, 200, 225, 250]

* 1. Create a dictionary where the keys are the first 5 letters of the alphabet and the values are their corre­ sponding positions (e.g., {'a' : 1, 'b' : 2, ... }). Swap the keys and values so that the letters become values and the numbers become keys. Write a snippet to perform the swap automatically. ***Hint:*** *You may want to consider dictionary comprehension.*
  2. The following dataset represents weekly sales figures for a company. Calculate the average sales.



weekly\_sales [200, 300, 250, 400, 350, 380, 420]



Reminder:

l *n*

*X* = - *LXi*

*n* i=l

The dataset below represents stock prices over 10 days. Calculate the (sample) standard deviation of the stock prices. You must use list comprehension. Feel free to use functions from the math module.

Reminder:

*s=*

Consider the following two sets:

set a {1, 2, 3, 4, 5}

set b {4, 5, 6, 7, 8}



[100, 102, 98, 105, 107, 106, 103, 110, 108, 107]

--

l *n*

*n* - l i=l

*I:(xi* - x)2

Perform and print the following operations: union, intersection, and difference between the two sets. What is the result when you check whether set\_a is a subset of set\_b?

* 1. Create a frozenset from a list of numbers [10, 20, 30, 40, 50]. Try adding a new number to this frozenset. What happens? Why can't you modify it?
  2. Consider the following snippet containing two small datasets:



prices\_A [100, 102, 98, 105, 107, 106, 103, 110, 108, 107]

prices\_B [101, 103, 99, 106, 108, 105, 102, 111, 109, 108]

These two lists represent the stock closing prices of two companies over a period of 10 days. Using list comprehension, determine on which days company B had a higher closing price than company A.

* 1. Create a list of numbers from 1 to 50. Using list comprehension, create a new list that contains the squares of all odd numbers.
  2. It looks like both of the following snippets delete list elements. What is the difference between these two snippets, if any?

sample\_list = [10, 20, 30] sample\_list[:] = [] print(sample\_list)

sample\_list = [10, 20, 30] sample\_list = [] print(sample\_list)

**Hint:** Use the id function before and after modifying the list in each snippet to see the difference.

* 1. Create a dictionary where the keys are names of 3 students and the values are another dictionary containing their scores in three subjects: Data Analytics, Buisness Analytics, and Python for Data Analysis. Access and print the Business Analytics score of the second student. Then, update the Python for Data Analysis score of the first student.



* 1. A warehouse manager at a factory monitors the daily demand for a certain tool. Based on historical data, the daily demand, *X,* can take any value between O and 9 (inclusive) with the following probabilities:

###### probability mass function (pmf)

demand= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

probabilities= [0.1, 0.05, 0.07, 0.01, 0.08, 0.12, 0.15, 0.09, 0.14, 0.09]

* + 1. Use zip O to combine the demand and probabilities lists into a dictionary.
    2. Using dictionary comprehension, calculate the expected daily demand and variance for the tool.
    3. The factory incurs a cost of $100 each time the tool is used. Define *Y* = lO0X, as the daily cost. Find the mean and variance of the daily cost *Y.*

Remember that

E(X) = *LXi. P(*=*X*Xi) and Var(X) = E(X2) - (E(X))2

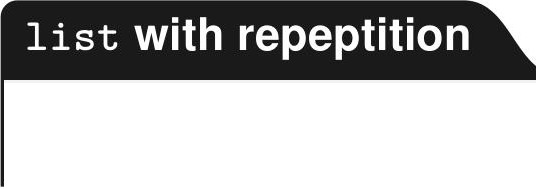
* 1. Create a dictionary with some names (as strings) as keys and ages as values. Try adding a new key that is the same name but with a different case (e.g., "Anna" and "anna"). What happens? Does the dictionary treat them as different keys?
  2. Consider the following snippet



str 1 = "Success is the result of consistent, persistent effort."

Use a dictionary to count how many times each character appears in this str. Ignore spaces and make sure the count is case-insensitive.

* 1. Consider the following list:



list\_nums = [1, 2, 1, 3, 5, 7, 7, 6, 3, 10, 7, 2, 1]

Write a snippet that returns a new list with only the unique elements from the above list. Do this in two different ways: with and without set O function.

* 1. Consider the following data



employees= {'Emily': 48\_000, 'Andrew': 52\_000, 'Mike': 47\_000, 'Anna': 60\_000}

Write a snippet that increases the salary of every employee by 10%, but only if their salary is below $50\_000. Print the updated dictionary of employees and their salaries.

* 1. An external audit of a company has revealed the following annual salaries (in thousands of dollars) for employees of an organization.



salaries= [50, 40, 500, 50, 40, 50, 40, 40, 80, 40]

Do you notice a significant difference between the mean and median salary values? Why do you think this difference exists?

**Hint:** A common way to find a median is to sort the data first and then use the following formula in the sorted data:

Median= {

Middle value,

½ x (Sum of two middle values),

if *n* is odd

if *n* is even

* 1. A shoe store wants to store the most common shoe sizes to meet customer demand. The following list

contains the shoe sizes sold in a retail store over the past month.



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| shoe sizes = [ | | | |  | | |
| 10, 9, 7, 6, 11, 8, 7, 9, 10, 8, 7, 12, 6, 9, 10, 9, 10, | | | | 10, 8, 7, | | |
| 7, 6, 8, 9, 7, 11, 6, 6, 9, 8, 7, 9, 8, 10, 11, 7, 7, 6, | | | | 12, 10, | | |
| 8, | 7, 10, 11, | 9, 7, 8, 8, 7, 6, 12, 8, 10, 6, 10, | 11, 9, | 7, | 6, | 7, |
| 8, | 10, 9, 12, | 11, 6, 10, 6, 9, 7, 9, 7, 8, 10, 7, | 9, 7, | 10, | 8, | 8, |
| 7, | 8, 9, 11, | 12, 9, 10, 8, 11, 9, 6, 10, 7, 6, 7, | 6, 10, | 8, | 9, | 12 |

1. Find mean, median, and mode of the sold shoe sizes.
2. Determine which metric (mean, median, or mode) is the most appropriate for answering the store manager's question.

Consider the following snippet showing the closing stock prices of three companies over several days:

**stock price datasets**

# *nvidia prices*

nvid\_p = [113.37, 117.87, 116.0, 116.26, 120.87, 123.51, 124.04,

121.4, 121.44, 117.0, 118.85, 122.85, 124.92, 127.72,

132.89, 132.65, 134.81, 134.8, 138.07, 131.6, 135.72, 136.93]

# *apple prices*

appl\_p = [220.69, 228.87, 228.2, 226.47, 227.37, 226.37, 227.52, 227.79,

233.0, 226.21, 226.78, 225.67, 226.8, 221.69, 225.77, 229.54,

229.04, 227.55, 231.3, 233.85, 231.78, 232.15]

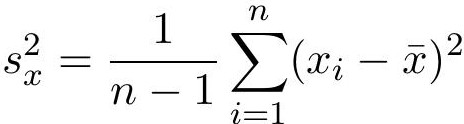
# *microsoft prices*

msft\_p = [430.81, 438.69, 435.27, 433.51, 429.17, 432.11, 431.31,

428.02, 430.3, 420.69, 417.13, 416.54, 416.06, 409.54,

414.71, 417.46, 415.84, 416.32, 419.14, 418.74, 416.12, 416.72]

* + 1. Using list comprehension, calculate the sample covariance between nvidia and microsoft stock prices.
    2. Using list comprehension, calculate the sample variance of apple stock prices.
    3. Using list comprehension, calculate the sample covariance of apple stock prices with itself.
    4. What is the relationship between the results from parts b and c? Does the outcome surprise you? Reminder: *sxy* is the sample covariance and s2 is the sample variance, calculated as:

1 *n*

Cov(x, y) = *sxy* = -- I)xi - *x)(Yi* - y)

*n* - 1 i=l

and

* 1. In the previous problem, using list comprehension:
     1. Calculate the Pearson correlation coefficient between nvidia and microsoft stock prices.
     2. Calculate the Pearson correlation coefficient of apple stock prices with itself. Reminder: *rxy* is the sample correlation coefficient, calculated as:

Corr(x, y) = *rxy*

*Sxy*

= --

*5x5y*

* 1. Consider the following dictionaries

###### kids favorite color

kid\_color 1 {'Locas': 'red', 'Ava': 'green', 'Mia': 'orange'}

kid\_color 2 {'Evelyn': 'black', 'Mia': 'white', }

Write a program that merges these two dictionaries. How many different ways do you know to merge these dictionaries? Print the resulting dictionary.

* 1. Consider the following dictionaries



sales\_q\_1 {'tv': 100, 'xbox': 200, 'macbook': 30}

sales\_q\_2 {'xbox': 150, 'speaker': 5}

Write a snippet that merges these two dictionaries. If a key exists in both dictionaries, the value should be the sum of the two values.

**Hint:** this is a challenging question; you may want to consider if.. else block inside a for loop.

* 1. Consider the following dictionary with students names and grades:



students = {'Noah': 90, 'Jack': 82, 'Sophia': 97, 'Lily': 92, 'Chloe': 77}

Write a snippet that returns a list of students who have grades above 85.

* 1. **A Mini Case Study** You are a data analyst working at a financial services company. Your task is to handle a small dataset that represents quarterly financial data for three companies. Below is a table representing the quarterly revenue and expenses for these companies:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Company** | **01 Revenue** | **01 Expenses** | **02 Revenue** | **02 Expenses** |
| Alpha | 120,000 | 80,000 | 140,000 | 90,000 |
| Beta | 200,000 | 150,000 | 220,000 | 160,000 |
| Gamma | 300,000 | 250,000 | 320,000 | 240,000 |

(a) Represent this data as a list of dictionaries. Each dictionary should have the company name as a key and a sub-dictionary as its value, where the sub-dictionary stores the quarterly revenue and expenses data. The structure should look like this:



[

{

'Company': 'Alpha', 'Financials': {

'Q1 Revenue': 120000,

'Q1 Expenses': 80000,

'Q2 Revenue': 140000,

'Q2 Expenses': 90000

}

},

]

(b)

Access and print the 02 revenue of the company **Beta.**

Access and print the total expenses for **Gamma** across both quarters (sum of 01 and 02 expenses). Calculate and print the net profit (Revenue - Expenses) for each company in **01.**

Update the **02 expenses of Alpha** to 95,000.

Add a new field called **01 Profit Margin** for each company in the dictionary. This should be calculated as:

Prof·tM

I argIn =

.

(01 Revenue - 01 Expenses)

0 1R evenue

x 100

Update the list of dictionaries to include this value for each company.

A new company, **Delta,** has just been added to your dataset. The company has the following financials:

* 01 Revenue: 180,000
* 01 Expenses: 130,000
* 02 Revenue: 190,000
* 02 Expenses: 140,000

Add **Delta** to your list of dictionaries and print the updated list.

Calculate the total revenue and total expenses across both quarters for all companies.

**sample output**

{'Total Revenue': total\_revenue\_value, 'Total Expenses': total\_expenses\_value

}

(c)

(d)

(e)

(f)

(g)

(h)

Print the summary report.

## Exercise Solutions

Solutions to these problems can be found on the following GitHub page: https://github.com/NaserNikandish/Python\_For\_Data\_Analysis You can also access the same link using the QR code below:



### References

**References and Resources**

The following references and resources were used in the preparation of these materials:

* 1. Official Python website at https: / /[www.python.org/.](http://www.python.org/)
  2. *Introduction to Computation and Programming Using Python,* John Guttag, The MIT Press, 2nd edition, 2016.
  3. *Python for Data Science Handbook: Essential Tools for Working with Data,* Jake VanderPlas, O'Reilly Media, 1st edition, 2016.
  4. *Data Mining for Business Analytics,* Galit Shmueli, Peter C. Bruce, Peter Gedeck, Nitin R. Patel, Wiley, 2020.
  5. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and /Python,* Wes McKinney, O'Reilly Media, 2nd edition, 2017.
  6. *Introduction to Python for Computer Science and Data Science,* Paul J. Deitel, Harvey Deitel, Pear­ son, 1st edition, 2019.
  7. *Data Visualization in Python with Pandas and Matplotlib,* David Landup, Independently published, 2021.
  8. *Introduction to Programming Using Java,* available at http:/ /math. hws. edu/ javanotes/.
  9. *Python for Programmers with Introductory Al Case Studies,* Paul Deitel, Harvey Deitel, Pearson, 1st edition, 2019.
  10. *Effective Pandas: Patterns for Data Manipulation (Treading on Python),* Matt Harrison, Independently published, 2021.
  11. *Introduction to Programming in Python; An Interdisciplinary Approach,* Robert Sedgewick, Kevin Wayne, Robert Dondero, Pearson, 1st edition, 2015.
  12. Python tutorials at https: / /betterprogramming. pub/.
  13. Python learning platform at https: / /[www.learnpython.org/.](http://www.learnpython.org/)
  14. Python resources at https: / /real python. com/.
  15. Python courses and tutorials at https: / /[www.datacamp.com/.](http://www.datacamp.com/)