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 `dict`, `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:
  - `dict`: 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.

[1] These 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

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 `dict`.

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 `[]`. 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`

```
list_1 = []                                                    # an empty list
list_2 = [5, 10, 15, 20, 30, 100]                              # homogeneous data
list_3 = ['python', 'Java', 'Data Science', 'I feel good']     # homogeneous data
list_4 = ["I'm in love with ", 'python ', 3.12]                 # heterogeneous data
```

2.1 Built-in `len()`, `type()`, `max()`, `min()`, and `sum()` Functions with a `list`

Python provides several built-in functions that make working with lists[2] easy and efficient. Let's explore some of these:

- `len()` returns the number of items in a `list`.
- `type()` tells you the data type of an object.
- `max()` 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}')
```

[2] These built-in functions work with other collection data types such as `tuple`, `dict`, `set`, and `range`.

```
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

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.

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 `0`, so the first element is at index `0`, 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]}')
```

```
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
```

2.3 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 `0`, which means the first element is at index `0`, 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.

2.4 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.

2.5 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 `[]` and gradually add elements using the `append()` method[3]. `append()` 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:

[3] A 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.

2.6 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 `[]`, making it a `list`:

Correct Use of +=

```
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 = []             # empty list
for item in range(10):
    my_numbers += [item]  # Wrap each item in a list before adding
print(my_numbers)
```

yields

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

Remember that as an alternative, you could use `append()` 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 = []                                  # empty list
my_list_1 += 'python'                           # treated as list
print(f'Output using +=: {my_list_1}')          # Output: ['p', 'y', 't', 'h', 'o', 'n']

my_list_2 = []                                  # empty list
my_list_2.append('python')                      # treated as string
print(f'Output using append(): {my_list_2}')    # 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']`. In the second case, `my_list_2.append('python')` adds the entire string `'python'` as a single element, resulting in `['python']`.

2.7 Creating a `list` Using the Built-in `list()` Function

In addition to the `append()` method and the `+=` operator, the built-in `list()` function provides another way to create lists. This function is particularly useful for converting various types of iterables — such as strings, ranges, and tuples[4] — into a `list`.

Here are some examples of using `list()`:

Converting a `string` into a `list()`

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

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

[4] To 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`.

`list()` Function

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

2.8 List Creation Using List Comprehension

So far, we have seen how to create a `list` using the `append()` method, the `+=` operator, and the `list()` 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]      # initial list
even_nums = []                                 # an empty list
for item in initial_list:
    if item % 2 == 0:                          # check even-ness
        even_nums.append(item)
print(even_nums)
```

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 `[]`:

With `list` Comprehension

```
initial_list = [2, 5, 7, 19, 20, 41, 50]                    # initial list
even_nums = [item for item in initial_list if item % 2 == 0] # list comprehension
print(even_nums)
```

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 item in iterable if condition == True]
```

Notice that the new list is enclosed inside a pair of square brackets `[]`. point to be noted.

Let's see more examples:

`list` Comprehension

```
list_1 = [2, 3, 7]
list_1_squared = [item ** 2 for item in list_1]            # squares of numbers
print(list_1_squared)

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.

2.9 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 `item` to the end of the list.
`insert()`	`list.insert(index, item)`	Inserts `item` 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 `item` in the list.
`index()`	`list.index(item)`	Returns the index of the first occurrence of `item`. Raises `ValueError` if not found.
`remove()`	`list.remove(item)`	Removes the first occurrence of `item`. 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

`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, 15]
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]:  3
after removal of first 3:  [5, 4, 7, 9, 3, 11, 13, 15]
after pop:  [5, 4, 7, 9, 3, 11, 13], Popped item:  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`	`item in list`	Checks if `item` exists in the list. Returns `True` if it exists.
`not in`	`item not in list`	Checks if `item` 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` `n` times 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 = [1, 2, 3]
list2 = [4, 5, 6]

print(f"Is 2 in list1? {'Yes' if 2 in list1 else 'No'}")          # Yes
print(f"Is 7 in list1? {'Yes' if 7 in list1 else 'No'}")          # 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 and list2: {new_list}")    # [1, 2, 3, 4, 5, 6]

repeated_list = list1 * 2                             # '*' for Repetition
print(f"Repeating list1 twice: {repeated_list}")      # [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 and list2:  [1, 2, 3, 4, 5, 6]
Repeating list1 twice:  [1, 2, 3, 1, 2, 3]
After deleting index 1 from list1:  [1, 3]
```

2.10 Difference Between `clear()` and `del`

The `clear()` 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()` Method on a `list`

    ```
    my_list = [1, 2, 3]
    my_list.clear()
    print(my_list)
    ```

    yields the empty list of `[]`. After using `clear()`, `my_list` is still a valid, empty list.
- `del`:
  - `del` is 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`

    ```
    my_list = [1, 2, 3]
    del my_list[1]    # Removes the element at index 1 (value 2)
    print(my_list)    # Output: [1, 3]
    ```

[5] and some other data structures like dictionaries, covered later
[6] In other words, the list object itself remains
[7] We 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
```

2.11 `sort()` Method Versus the Built-in `sorted()` Function

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

- `sort()` is a `list` method that modifies the original list **in-place**[8]. 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'Original list: {grades}')       # The original list remains unchanged
print(f'Sorted list: {sorted_grades}')  # A new list is created with sorted values
```

This yields:

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

[8] You'll 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 as `reverse=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.

2.12 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, 2, 3], [4, 5, 6], [7, 8]]
print(two_dim_list)
```

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:

Accessing Rows

```
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:

[9] You 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[0])  # 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[0][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`.

3 Sequence Data Types: `tuple`

A `tuple` consists of values separated by commas `,` and is enclosed in (optional) parentheses `()`.

Here are a few examples:

Creating `tuple`

```
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 `()` are optional, but including them is recommended for better code readability. 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.

3.1 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 `[]`. 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
```

3.2 Using += with a `tuple`

Item assignment using square brackets `[]` 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 reference 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.

3.3 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()` 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.

3.4 `tuple()` Function

We have previously seen type conversion functions like `int()`, `float()`, `str()`, and `list()` to convert values into integers, floats, strings, and lists, respectively. Similarly, Python provides `tuple()` 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()` and `tuple()`functions are useful when you need to change the type of an iterable for further operations. 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()` function

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

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

`list` slicing

```
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
```

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[:]` creates a *shallow* copy of the list. You can use the `weekly_sales.copy()` method, alternatively. 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:

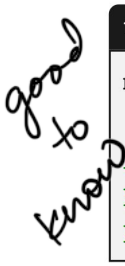
Removing elements using slicing

```
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:



`tuple` slicing

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

print(f'months[2:5] is {months[2:5]}')        # elements from index 2 to 4
print(f'months[:3] is {months[:3]}')          # elements from start to index 2
print(f'months[::2] is {months[::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

```python
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, 0]
```

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

Sequence Slicing

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]`.

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`

```python
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]                             # Packing a list
stats_grade, python_grade = my_grades             # Unpacking the list
print(python_grade)
```

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]                              # Packing a list
stats_grade, python_grade = my_grades                  # Unpacking the list
print(python_grade)
```

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:

Unpacking with `*`

```
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 `[]`
- 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.

6 The `zip()` & `enumerate()` Functions

6.1 `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:

6.1.1 Using a `for` loop with `range()` (Less Pythonic)

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

Iterating with indices using `range()`

```
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
```

6.1.2 Using `enumerate()` 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()` function. This function takes an iterable and gives you both the index and value for each item.

If you pass `enumerate()` into the `list()` 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()`

```
test_1 = [20, 30, 40, 50]
for index, value in enumerate(test_1):
    print(index, value)
```

iterable tuples of (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()`

```
test_1 = [20, 30, 40, 50]
for index, value in enumerate(test_1, start=100):
    print(index, value)
```

starts with index = 100.

This yields:

```
100 20
101 30
102 40
103 50
```

`enumerate()` Function

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.

6.2 `zip()` 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()` 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()` function to convert the output of the `zip()` function into a `list`.

Here's an example:

Using `zip()` Function

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

wedding_pairs = list(zip(brides, grooms))
print(wedding_pairs)
```

This yields:

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

In this example, the `zip()` 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()` to pair more than two lists:

Using `zip()` 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.

7 Generator Expressions

7.1 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 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 `()`, whereas a list comprehension uses square brackets `[]`.

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 0x000002719E3030B0>
```

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
```

2. **Convert the generator to a list using the `list()` function**

Convert Generator to List

```
generator_expression = (x ** 2 for x in range(4))
print(list(generator_expression))
```

This yields:

```
[0, 1, 4, 9]
```

7.2 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 = '''
list_comp = [item for item in range(1_000_000) if item % 2 == 0]
'''
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)
'''
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.

7.3 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 0 to 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, generator expressions generate elements one at a time needing less computational resources. For memory- or performance-sensitive applications, consider using generator expressions.

7.4 Tuple Comprehension: Is It Possible?

Generator expressions may bring to mind the idea of tuple comprehensions since they are enclosed in parentheses `()`. 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 ( )

```
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()` function.

Here's how:

Using `tuple()` 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(5))
print(tuple_1)
```

This yields:

```
(0, 1, 2, 3, 4)
```

We've now created a tuple using the `tuple()` 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()` 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.

8 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:

- `dict` (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 `dict` and `set` data types.

9 Non-Sequence Data Type: `dict`

A `dictionary`[10] 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 `dict`, checking its type and length

```
superhero_dict = {}                    # an empty dictionary
superhero_dict['name'] = 'Thor'        # Assign value of 'Thor' to 'name' key
superhero_dict['strength'] = 95        # Assign value of 95 to 'strength' key

superhero_dict['abilities'] = ['lightning', 'hammer', 'flight']
```

[10] Dictionaries are 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 'dict'>
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 `dict`

```
superhero_dict['strength'] = 98      # Update existing key-value pair
superhero_dict['intelligence'] = 85 # Add a new key-value pair
print(superhero_dict)
```

This yields:

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

Besides `len()` and `type()`, other built-in functions like `sorted()` can also be used with dictionaries. By default, these functions operate on the dictionary's keys[11].

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))           # number of key-value pairs = 3
print(sorted(superhero_strengths))        # 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
```

[11] Unless explicitly specified to apply to the values.

Dictionary `dict`

A **dictionary** (`dict`) 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.

9.1 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 methods[12] and operations:

Method	Sample Syntax	Description
`get()`	`dict.get(key, default)`	Returns value for `key`, or `default` if not found.
`keys()`	`dict.keys()`	Returns a view object of all the keys in the dictionary.
`values()`	`dict.values()`	Returns a view object of all the values in the dictionary.
`items()`	`dict.items()`	Returns a view object of key-value pairs as tuples.
`copy()`	`dict.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()`	`dict.clear()`	Removes all elements from the dictionary.

Table 4: Important Dictionary Methods

9.2 `get()` Method for Accessing `dict` Values

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

Using `get()` Method

```
Dog_dict = {'name': 'Milo',
            'age': 9,
            'surgeries': ['jaw', 'l-leg', 'r-leg']
            }
print(Dog_dict.get('age'))                    # Returns 9
print(Dog_dict.get('Vet', 'Key not found'))   # Returns 'Key not found'
```

[12] There are less used methods such as `fromkeys()`, `pop`, `popitem`, and `setdefault` and you are encouraged to explore them yourself.

Operator/Keyword	Sample Syntax	Description
`in`	`key in dict`	Returns `True` if the key is in the dictionary.
`not in`	`key not in dict`	Returns `True` if the key is not in the dictionary.
`==`	`dict1 == dict2`	Returns `True` if two dictionaries have the same key-value pairs.
`!=`	`dict1 != dict2`	Returns `True` if two dictionaries are not equal.
`del`	`del dict[key]` `del dict`	Deletes the specified key-value pair from the dictionary. Can also delete the entire dictionary.
`|`	`dict1 | dict2`	Returns a new dictionary by merging two dictionaries.
`**`	`dict1 = {**dict2, **dict3}`	Merges two dictionaries using unpacking.
`*`	`*dict`	Unpacks keys of the dictionary. Used inside a function such as `print(*dict)`

Table 5: Important Dictionary Operators and Keywords

This yields:

```
9
Key not found
```

9.3 `keys()` Method

The `keys()` method returns a view object[13] that displays all the keys in the dictionary. This allows you to loop through the keys directly, as shown in the example below:

Using `keys()` Method

```python
for k in Dog_dict.keys():
    print(k)
```

This yields:

```
name
age
surgeries
```

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

[13] A 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()` method

```
Dog_dict = {'name': 'Milo', 'age': 9}
Dog_keys = Dog_dict.keys()
Dog_keys[0] = '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()` into a list using the `list()` 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']
```

9.4 `values()` Method

The `values()` 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()` method, the values cannot be modified through the `values()` method because it only provides a view of the current values.

Using `values()` Method

```
Dog_dict = {'name': 'Milo',
            'age': 9,
            'surgeries': ['jaw', 'l-leg', 'r-leg']
            }
for v in Dog_dict.values():
    print(v)
```

This yields:

```
'Milo'
9
['jaw', 'l-leg', 'r-leg']
```

Just like with the `keys()` method, you can use the `list()` 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', 'l-leg', 'r-leg']]
```

9.5 `items()` Method

The `items()` 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 `items()` Method

```
for k, v in Dog_dict.items():
    print(f'Key = {k}, Value = {v}')
```

This yields:

```
Key = name, Value = Milo
Key = age, Value = 9
Key = surgeries, Value = ['jaw', 'l-leg', 'r-leg']
```

In this example, the `items()` 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, respectively. 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 `items()`, 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()` function:

Converting items to a list of tuples

```
items_list = list(Dog_dict.items())  # Convert items to a list of tuples
print(items_list)
```

This yields:

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

9.6 `copy()` Method

The `copy()` method creates a shallow copy of the dictionary.[14]

Using `copy()` Method

```
Dog_dict = {'name': 'Milo', 'age': 9, 'surgeries': ['jaw', 'l-leg', 'r-leg']}
copy_dict = Dog_dict.copy()  # Create a shallow copy
print(copy_dict)
```

This yields:

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

9.7 `update()` Method

The `update()` 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()` 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: Imp point

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

The `update()` 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.

good to Know

Using `update()` with an Iterable

```
dict_E = {'BARM': 100, 'Marketing': 150}
my_list = [('Finance', 120), ('HCM', 40)]  # List of tuples
dict_E.update(my_list)
print(dict_E)
```

[14] This 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.

[15] You 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 `|` (Union Operator)

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

Merging Dictionaries with `|`

```
dict_A = {'Python': 100, 'Simulation': 99}
dict_B = {'BA': 95, 'Python': 90}
dict_C = {'Statistics': 85, 'OM': 70, 'Python': 9000}

dict_D = dict_A | dict_B | dict_C
print(dict_D)
```

This yields:

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

You can merge more than two dictionaries using `|`. 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 `**`

```
dict_A = {'Python': 100, 'Simulation': 99}
dict_B = {'BA': 95, 'Python': 90}
dict_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}
```

9.8 `clear()` Method

The `clear()` 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()` method

```
test_dict = {'k1': 10, 'k2': 20}
test_dict.clear()
print(test_dict)
```

This yields:

```
{}
```

9.9 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', 'l-leg', 'r-leg']}

print('name' in Dog_dict)        # Check if 'name' is a key
print('Vet' in Dog_dict)         # Check if 'Vet' is a key
print('Vet' not in Dog_dict)     # Check if 'Vet' is not a key
```

This yields:

```
True
False
True
```

9.10 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

```
dict_A = {'name': 'Milo', 'age': 9}
dict_B = {'age': 9, 'name': 'Milo'}
dict_C = {'name': 'Milo', 'age': 9, 'surgeries': ['jaw', 'l-leg', 'r-leg']}

print(dict_A == dict_B)  # Should be True
```

```
print(dict_B != dict_C)  # Should be True
```

This yields:

```
True
True
```

9.11 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', 'l-leg', 'r-leg']}

del Dog_dict['age']    # Removes the key 'age' and its value
print(Dog_dict)
```

This yields:

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

del Dog_dict            # Completely removes Dog_dict
print(Dog_dict)
```

This yields:

```
NameError: name 'Dog_dict' is not defined
```

9.12 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
```

9.13 `dict()` 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 `dict()` 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 `dict()` to build dictionaries from key-value pairs or iterable objects such as lists of tuples, as shown in the below example:

Using `dict()` function

```
dict_1 = dict(name = "James", age = 25)  # Creating a dict with key-value pairs
print(dict_1)

list_of_tuples = [('x', 1), ('y', 2), ('z', 3)]
dict_2 = dict(list_of_tuples)            # Creating a dict 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()` function. This function combines two lists or other iterables into pairs, and then `dict()` can turn these pairs into key-value pairs for the dictionary. Here's an example:

Creating a dictionary with `zip()`

```
keys = ['name', 'age', 'job']
values = ['Milo', 9, 'Chief Barking Officer']
new_dict = dict(zip(keys, values))
print(new_dict)
```

good usecase of zip function.

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 `dict()` function.

9.14 Dictionary Comprehension

So far, we have seen several ways of creating dictionaries, such as manually assigning key-value pairs, using the `update()` method, leveraging the `zip()` function, and utilizing the `dict()` function. However, **dictionary comprehension** 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.items() if condition}
```

This structure allows you to conditionally include items from `old_dict` in the new dictionary, while also transforming 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:

Modifying values

```
hero_stats = {"John Wick": 85, "Ethan Hunt": 90, "Lara Croft": 75, "Jason Bourne": 80}
boosted_stats = {k: v * 2 for k, v in hero_stats.items()}
print(boosted_stats)
```

This yields:

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

9.15 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}
```

9.16 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 for k, 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
```

9.17 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 for k, v in hero_stats.items() if v > 80}
print(elite_heroes)
```

This yields:

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

9.18 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 `0`; 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`.

[16] There 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 for k, v in dict_A.items() if v is not None}
print(dict_B)
```

This yields:

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

9.19 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 {dict_A}')
print(f'dict_B is {dict_B}')
```

This yields:

```
dict_A is {}
dict_B is {}
```

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

9.20 Two Ways to Copy a Dictionary

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

- Using the `dict()` function
- Using the `copy()` method

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

more at https://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 {dict_A}')
print(f'dict_B is {dict_B}')
```

This yields:

```
dict_A is {}
dict_B is {'Python': 95, 'Statistics': 100}
```

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

9.21 Nested Dictionaries

A nested dictionary is a dictionary that contains other dictionaries as its values. This structure allows you to organize 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.

10 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.

11 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:

Shallow Copy

```
original = {'a': 1, 'b': [2, 3, 4]}
shallow_copy = original.copy()

shallow_copy['b'][0] = 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 `deepcopy()` 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()` and `deepcopy()`, you can refer to the official Python documentation: https://docs.python.org/3/library/copy.html.

`dict()` Function

The **dict()** 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.

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, `{}`.

Set Creation

```
games_A = {'Mario', 'Zelda', 'Sonic'} # no repetition
games_B = {'FIFA', 'Halo', 'FIFA', 'Minecraft', 'Mario', 'Halo'} # with repetition

print(games_A)
print(games_B)
```

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, `[]`. 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.

12.1 `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 `item` to the set
`update()`	`set.update(iterable)`	Add multiple elements from `iterable`
`remove()`	`set.remove(item)`	Remove `item` from the set. Raises an error if `item` is not found
`discard()`	`set.discard(item)`	Remove `item` if it exists. Does not raise an error if `item` 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()
```

12.2 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
`|`	`set_A | 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 | Operator

```
superheroes = {'Iron Man', 'Captain America', 'Black Panther'}
video_games = {'Call of Duty', 'Fortnite', 'Black Panther'}
union_set = superheroes | 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
```

12.3 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()` 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}
```

set

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

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

`frozenset()` 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'
```

`frozenset`

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.

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	Iterable	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 `dict`	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, `dict` 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.

14 Exercises

1. What is the output of running this code?

What is the output?

```
list_1 = []
for item in range(3):
    list_1[item].append(item)
print(item)
print(list_1)
```

2. 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?
3. 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`?
4. 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?
5. Consider the following snippet

working with `str`

```
str_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?

6. Create a `range()` 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`?
7. Use the `range()` function to generate a sequence of numbers from 0 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?
8. You are given a list of daily sales figures for a week:

daily sales data

```
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 `itertools` functions). **Hint:** rely on `append()` method inside a `for` loop. The output `list` should be:

ideal output for `cumsum`

```
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:

ideal output for running average

```
running_avg_sales = [100, 125, 150, 175, 200, 225, 250]
```

9. Create a dictionary where the keys are the first 5 letters of the alphabet and the values are their corresponding 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.*
10. The following dataset represents weekly sales figures for a company. Calculate the average sales.

Weekly sales

```
weekly_sales = [200, 300, 250, 400, 350, 380, 420]
```

Reminder:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

11. 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.

stock prices

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

Reminder:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

12. Consider the following two sets:

`set` operations

```
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}
```

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`?

13. 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?
14. Consider the following snippet containing two small datasets:

price 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.

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

Snippet A

```
sample_list = [10, 20, 30]
sample_list[:] = []
print(sample_list)
```

Snippet B

```
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.

17. 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.
18. 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 0 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]
```

(a) Use `zip()` to combine the `demand` and `probabilities` lists into a dictionary.
(b) Using dictionary comprehension, calculate the expected daily demand and variance for the tool.
(c) The factory incurs a cost of $100 each time the tool is used. Define $Y = 100X$, as the daily cost. Find the mean and variance of the daily cost Y.
Remember that

$$E(X) = \sum_i x_i \cdot P(X = x_i) \qquad \text{and} \qquad \text{Var}(X) = E(X^2) - (E(X))^2$$

19. 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?
20. Consider the following snippet

Count chars

```
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.

21. Consider the following `list`:

`list` with repeptition

```
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()` function.

22. Consider the following data

Employee salaries

```
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.

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

salaries

```
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:

$$\text{Median} = \begin{cases} \text{Middle value}, & \text{if } n \text{ is odd} \\ \frac{1}{2} \times (\text{Sum of two middle values}), & \text{if } n \text{ is even} \end{cases}$$

24. 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.

sold shoe sizes

```
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
```

```
]
```

(a) Find mean, median, and mode of the sold shoe sizes.

(b) Determine which metric (mean, median, or mode) is the most appropriate for answering the store manager's question.

25. 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]
```

(a) Using list comprehension, calculate the sample covariance between `nvidia` and `microsoft` stock prices.

(b) Using list comprehension, calculate the sample variance of `apple` stock prices.

(c) Using list comprehension, calculate the sample covariance of `apple` stock prices with itself.

(d) What is the relationship between the results from parts b and c? Does the outcome surprise you? Reminder: s_{xy} is the sample covariance and s^2 is the sample variance, calculated as:

$$\text{Cov}(x,y) = s_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \qquad \text{and} \qquad s_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

26. In the previous problem, using list comprehension:

(a) Calculate the Pearson correlation coefficient between `nvidia` and `microsoft` stock prices.

(b) Calculate the Pearson correlation coefficient of `apple` stock prices with itself.
Reminder: r_{xy} is the sample correlation coefficient, calculated as:

$$\text{Corr}(x,y) = r_{xy} = \frac{s_{xy}}{s_x s_y}$$

27. 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.

28. Consider the following dictionaries

sales data

```
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.

29. Consider the following dictionary with students names and grades:

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.

30. **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	Q1 Revenue	Q1 Expenses	Q2 Revenue	Q2 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:

sample output

```
[
{
'Company': 'Alpha',
'Financials': {
'Q1 Revenue': 120000,
'Q1 Expenses': 80000,
'Q2 Revenue': 140000,
'Q2 Expenses': 90000
```

```
        }
    }, ...
    ]
```

(b) Access and print the Q2 revenue of the company **Beta**.

(c) Access and print the total expenses for **Gamma** across both quarters (sum of Q1 and Q2 expenses).

(d) Calculate and print the net profit (Revenue - Expenses) for each company in **Q1**.

(e) Update the **Q2 expenses** of **Alpha** to 95,000.

(f) Add a new field called **Q1 Profit Margin** for each company in the dictionary. This should be calculated as:

$$\text{Profit Margin} = \left(\frac{\text{Q1 Revenue} - \text{Q1 Expenses}}{\text{Q1 Revenue}}\right) \times 100$$

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

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

- Q1 Revenue: 180,000
- Q1 Expenses: 130,000
- Q2 Revenue: 190,000
- Q2 Expenses: 140,000

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

(h) 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
}
```

Print the summary report.

15 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:

16 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/.
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 IPython*, Wes McKinney, O'Reilly Media, 2nd edition, 2017.
6. *Introduction to Python for Computer Science and Data Science*, Paul J. Deitel, Harvey Deitel, Pearson, 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 AI 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/.
14. Python resources at https://realpython.com/.
15. Python courses and tutorials at https://www.datacamp.com/.