Sets

PF2

Learning Objectives

1. **Create** and **manipulate** sets in Python
2. Understand the unique **properties** of sets
3. Perform mathematical set **operations**
4. Identify **when to use sets** versus other data structures like lists and tuples

Introduction

Another built-in Python collection type is the **set**. Unlike lists and tuples, sets are **unordered** collections of **unique** items.

Let's break down those two characteristics:

- **Unordered**: Items in a set *do not have a defined order or index*; you cannot access elements by position
- **Unique**: A set *cannot contain duplicate* values; each element must be unique

These properties make sets particularly useful for **efficiently checking for membership** using the `in` operator, **removing duplicates** from other collections, and **performing mathematical set operations** that find the **common** or **different items** between sets.

See here for the documentation of `sets` from Python.org.

Creating sets

Sets can be created using **curly braces** `{}` or calling the `set()` class on a list or tuple. When creating a set with curly braces, duplicate values are automatically removed.

```
1 # Creating a set of unique sample identifiers
2 unique_samples = {'sample1', 'sample2', 'sample3', 'sample1', 'sampleC'}
```

```
print(unique_samples)
```

```
{'sampleC', 'sample2', 'sample1', 'sample3'}
```

Notice that `sample1` appears twice in our creation, but the set **automatically removes the duplicate**. This function helps make `set()` more than a data structure, but all a powerful *data cleaning tool*.

In the cell below, let's explore adding two equal values (the number '1'), but inserting them into our set as two different types (integer and string, respectively).

```
# Creating a set from a list to remove duplicates
samples = [1, 1, '1', 2, 5]
unique_samples = set(samples)
print(unique_samples)
```

```
{'1', 1, 2, 5}
```

NOTE

This example illustrates that Python distinguishes between `1` (an integer) and `'1'` (a string), so **both are preserved** as unique values in the set.

PRACTICE EXERCISE

1.

You have a list of gene names from a sequencing experiment that contains duplicates: `gene_list`. Create a set called `unique_genes` to remove the duplicates.

```
gene_list = ['BRCA1', 'TP53', 'BRCA1', 'MYC', 'TP53', 'KRAS', 'MYC']
# Create your unique_genes set here
```

Run Code

```
Ready to run code!
```

Creating an empty set

Using `{}` alone creates an empty dictionary (a different data structure we will learn about in Python Fundamentals 3), *not* a set.

The cell below demonstrates why you shouldn't use *empty* curly braces `{}` to instantiate a set:

```
object = {}

print(type(object))
```

```
<class 'dict'>
```

Using the `type()` class, we can confirm that empty curly braces create a dictionary.

However, as we have seen earlier in this lesson, when appropriately used, curly brackets `{}` will create a set.

However, as `set()` is usually deployed to clean a list or tuple (of duplicates), you will most often find yourself using the `set()` class, as follows:

```
# Creating an empty set
empty_set = set()

print(empty_set)
print(type(empty_set))
```

```
set()
<class 'set'>
```

Mutability

Like lists, sets are **mutable**, meaning that you can add or remove elements, after they are instantiated (created).

As with lists, set objects have their own built-in, associated methods. One of these is `.add`, allowing us to add new items to an existing set.

```
```

```python
# Instantiate a set:
gene_set = {'geneA', 'geneB', 'geneC'}
print("Original set:", gene_set)

# Add new elements to this set:

gene_set.add('geneD') # Use the .add() method
print("After adding geneD:", gene_set)
```

```
Original set: {'geneB', 'geneC', 'geneA'}
After adding geneD: {'geneB', 'geneC', 'geneA', 'geneD'}
```

If you attempt to add a duplicate, the operation is simply *ignored* without raising an error:

```python
gene_set.add('geneA')  # This already exists
print("After attempting to add geneA again:", gene_set)
```

```
After attempting to add geneA again: {'geneB', 'geneC', 'geneA', 'geneD'}
```

Other common set methods

Unique items in a set can be removed using `.remove()` or `.discard()`:

```python
gene_set.remove('geneB')                    # Use .remove() to remove a specified e
print("After removing geneB:", gene_set)

# .discard() won't raise an error if the element doesn't exist:
gene_set.discard('geneZ')                   # This doesn't exist, but no error

print("After attempting to discard geneZ:", gene_set)
```

```
After removing geneB: {'geneC', 'geneA', 'geneD'}
After attempting to discard geneZ: {'geneC', 'geneA', 'geneD'}
```

```python
# Practice adding and removing elements from sets
```

Run Code

```
Ready to run code!
```

PRACTICE EXERCISE

2.

Create an empty set called `patient_ids`. Add the following patient IDs to it, one by one:

- 'P001', 'P002', 'P003'.

Then check if 'P002' is in the set and print the result.

```
# Create and populate your patient_ids set here
```

Run Code

```
Ready to run code!
```

Indexing

Because sets are unordered, they **do not support indexing or slicing** like lists do. Attempting to access elements by position will raise a `TypeError`:

```
# Run this cell to see the error: TypeError
gene = gene_set[0]
```

Run Code

```
Ready to run code!
```

However, you can still check for an element's membership using the `in` operator:

```
print('geneA' in gene_set)
```

```
print('geneZ' in gene_set)
```

```
True
False
```

Set operations

Sets provide fast and expressive ways to *compare* collections. This is particularly powerful in biological data analysis, where we often need to compare different datasets to one another in order to find overlaps, or identify unique elements.

In the code cell below, we will use set operations to compare differentially-expressed genes from two experiments:

```
# Example: Comparing differentially expressed genes from two experiments
experiment1_genes = {'TP53', 'MYC', 'BRCA1', 'AKT1', 'KRAS'}
experiment2_genes = {'MYC', 'VEGFA', 'KRAS', 'MAPK1', 'TP53'}

print("Experiment 1 genes:", experiment1_genes)
print("Experiment 2 genes:", experiment2_genes)
```

```
Experiment 1 genes: {'KRAS', 'MYC', 'TP53', 'AKT1', 'BRCA1'}
Experiment 2 genes: {'VEGFA', 'KRAS', 'MYC', 'TP53', 'MAPK1'}
```

We can also make use of the `.union()` method to identify all elements *unique* to multiple sets. The method returns a brand new set, containing only unique elements. This can be used on two or more sets.

In this example, let's identify genes unique to each of the two experiments:

```
all_found_genes = experiment1_genes.union(experiment2_genes)

print(f"Unique genes:  {all_found_genes}")
print(f"\nTotal unique genes: {len(all_found_genes)}")
print(f"\nType of object returned by .union():    {type(all_found_genes)}")
```

```
Unique genes:  {'VEGFA', 'MYC', 'MAPK1', 'BRCA1', 'KRAS', 'TP53', 'AKT1'}

Total unique genes: 7
```

```
Type of object returned by .union():    <class 'set'>
```

The `.intersection()` method finds elements common to two or more sets.

In the code cell below, we are going to create a third set, with one gene that is common to the two sets created, previously.

```
experiment3_genes = {'EGFR', 'AKT1', 'PTEN', 'VEGFA', 'BRCA2', 'MAPK3', 'KRAS'} # L

common_genes = experiment1_genes.intersection(experiment2_genes, experiment3_genes)

print("Common genes:", common_genes)
print("\nNumber of overlapping genes:", len(common_genes))
print(f"\nType of object returned by .intersection():   {type(common_genes)}")
```

```
Common genes: {'KRAS'}

Number of overlapping genes: 1

Type of object returned by .intersection():   <class 'set'>
```

> **NOTE**
>
> For comparing multiple sets using these methods, the syntax is to separate them using **commas**, inside the round parentheses of the method being applied.

In order to find genes unique to one set only, we can use the set method `.difference()`. The set to which the method is appended, is the set in whose elements we are searching for unique items. As with all the methods discussed, a new set is returned containing the unique elements.

```
unique_to_exp1 = experiment1_genes.difference(experiment2_genes, experiment3_genes)

print("Unique to Experiment 1:", unique_to_exp1)
```

```
Unique to Experiment 1: {'BRCA1'}
```

`.symmetric_difference` and `^`

In order to return a new set containing either of the elements in **two** sets, but not in both, we can use `.symmetric_difference()`. Explicitly, this returns **everything *except* what is shared** between two sets.

```
exclusive_genes = experiment1_genes.symmetric_difference(experiment2_genes)
```

```
print("Genes found exclusively in one experiment:", exclusive_genes)
```

```
Genes found exclusively in one experiment: {'VEGFA', 'MAPK1', 'AKT1',
'BRCA1'}
```

> **NOTE**
>
> The `.symmetric_difference()` method operates on *only* two comparison sets. If you try to use three or more sets, a `TypeError` will be raised.

To get around this, you can also use the caret ^ operator to evaluate the same relationship between sets. Namely, **everything *except* what is shared** between sets.

However, ^ offers one main advantage. And that is that you can compare more than two sets. This is demonstrated in the code cell below.

```
exclusive_genes = experiment1_genes ^ experiment2_genes ^ experiment3_genes
print("Genes found exclusively in one experiment:", exclusive_genes)
```

```
Genes found exclusively in one experiment: {'MAPK1', 'EGFR', 'MAPK3',
'BRCA1', 'BRCA2', 'KRAS', 'PTEN'}
```

`set` operations and other mathematical operators As with `^`, you can also use other mathematical operators for different set operations.

```
# Alternative syntax using operators
print("Union using |:", experiment1_genes | experiment2_genes)
print("Intersection using &:", experiment1_genes & experiment2_genes)
print("Difference using -:", experiment1_genes - experiment2_genes)
print("Symmetric difference using ^:", experiment1_genes ^ experiment2_genes)
```

```
Union using |: {'VEGFA', 'MYC', 'MAPK1', 'BRCA1', 'KRAS', 'TP53', 'AKT1'}
Intersection using &: {'TP53', 'KRAS', 'MYC'}
Difference using -: {'BRCA1', 'AKT1'}
Symmetric difference using ^: {'VEGFA', 'MAPK1', 'AKT1', 'BRCA1'}
```

For clarity, this table summarises the different methods, operators and operations associated with them.

Set Operations in Python

Operation	Method syntax	Operator syntax	Additional info / unique behaviour			
Union	`set1.union(set2)`	`set1 | set2`	Combines all unique elements from both sets. Both forms allow comparison across multiple sets (e.g. `set1.union(set2, set3)` or `set1 | set2 | s`			
Intersection	`set1.intersection(set2)`	`set1 & set2`	Returns only elements present in *both* sets. Both forms support multiple sets (e.g. `set1 & set2 & set3`).			
Difference	`set1.difference(set2)`	`set1 - set2`	Returns elements found in the first set but **not** in the second. Both forms support chaining across multiple sets.			
Symmetric difference	`set1.symmetric_difference(set2)`	`set1 ^ set2`	Returns elements that are **in either set but not both**. The `.symmetric_difference()` method only works with two sets; the operator `^` can be **chained** to compare more than two (e.g. `set1 ^ set2 ^ set3`).			

```
# Practice with set operations using your own biological examples
```

Run Code

Ready to run code!

PRACTICE EXERCISE

3.

You're analysing antimicrobial resistance patterns in bacterial isolates. You have resistance genes found in two different patient populations:

```
population_A_resistance = {'blaTEM', 'aac', 'tetA', 'sul1', 'qnrS'}
population_B_resistance = {'blaCTX', 'aac', 'tetA', 'sul2', 'qnrS', 'mecA'}
```

1. Find all resistance genes present across both populations
2. Find resistance genes common to both populations
3. Find resistance genes unique to population A
4. Find resistance genes that appear in only one population (not both)

```python
population_A_resistance = {'blaTEM', 'aac', 'tetA', 'sul1', 'qnrS'}
population_B_resistance = {'blaCTX', 'aac', 'tetA', 'sul2', 'qnrS', 'mecA'}

# Complete the analysis here
```

Run Code

```
Ready to run code!
```

Summary

In this section of Python Fundamentals 2, we explored Python **sets** — collections of unique, unordered elements that are ideal for comparing, combining and filtering data. We learned how to create sets, add and remove items, and perform key operations such as **union**, **intersection**, **difference**, and **symmetric difference**. We also examined both the **method syntax** (e.g. `.union()`, `.intersection()`) and the **operator syntax** (e.g. `|`, `&`, `-`, `^`), noting subtle differences in how they behave. Together, these tools make sets a powerful structure for handling distinct data and performing fast membership or comparison tasks in Python.