Introduction

PF2

Python is all about data (at least for researchers), so with that in mind, we will be exploring some of the core data structures in Python.

Previously, you will have learnt about using variables to store individual data points, i.e. integers, floats, or strings. Here, we'll build upon that by looking at how you can store multiple data points in a single structure, specifically **lists**, **tuples**, and **sets**. There are several common features amongst them all; however, each data structure has its own set of unique features, often restricting what data goes which can facilitate future operations.

Primed with this new knowledge, we'll revisit **strings** and look at them in a new light. Not as a simple store of words, but as sequential stores of information themselves.

Native Python Data Structures

In Python there are four built-in (no need for importing - see **PF3**) data structures that can store collections of any data points. These are **lists**, **tuples**, **sets**, and **dictionaries**. We'll cover the first three in this lesson with **dictionaries** covered in the next lesson **PF3**. Each structure has a unique way and rules for storing data, with lists being the most general. One aspect that does tie them all together, is that they can all store any type of data and can be iterated (or looped) through. This is the process of accessing each element or data point in the data structure one at a time, traversing their contents sequentially, this will be also be covered in **PF3**. This is also possible with **strings**, which is why we have chosen to include them in this lesson. However, as **strings** are limited to handling text/character data, they can be viewed as a specialised data structure.

Learning Overview

Concepts	Synergy	Syntax (Functions & Methods)	Objectives	Common Pitfalls / Notes
Data Structures & Collections	**PF1**: Variables and data types provide foundation for understanding how data is stored	`[]`, `()`, `{}`, `""` - Creation syntax `len()` - Length function `type()` - Type checking	By the end of this lesson, you should be able to create and differentiate between Python's native data structures	Empty collections can be created with `[]` or `list()`, but `{}` creates an empty dict, not a set - use `set()` instead
Mutability vs Immutability	**PF3**: Dictionaries extend mutable collections	Lists: `.append()`, `.remove()`, `.pop()`, `.insert()` Tuples: Immutable - no modification methods Sets: `.add()`, `.remove()`, `.discard()`	Understand when data can be modified after creation and choose appropriate data structures based on mutability requirements	Strings are immutable despite appearing modifiable - methods like `.replace()` return new strings.
Indexing & Slicing	**Data Handling modules**: Array indexing in NumPy/Pandas	`[0]`, `[-1]` - Index access `[start:end]` - Slicing syntax `.index()` - Find position of value	Access and extract specific elements or ranges from ordered collections using Python indexing conventions	End index in slicing is exclusive: `[0:3]` returns indices 0, 1, 2.
Collection Operations	**PF3**: Iteration and loops work with all collections **Stats modules**: Data manipulation and analysis	Lists: `+` concatenation, `.extend()`, `.sort()` Sets: `.union()`, `.intersection()`, `.difference()` Strings: `+` concatenation, multiplication	Perform mathematical and logical operations between collections, including set theory operations	`.sort()` modifies in place and returns None; `sorted()` returns new sorted list. Set operations lose order.
Membership Testing	**PF1**: Boolean logic and conditional statements **Data validation**: Checking data integrity	`in` and `not in` operators `.count()` - Frequency counting Set operations for efficient membership	Test whether elements exist in collections and validate data presence efficiently	ADD SOMETHING
String Manipulation	**PF4**: Text parsing and cleaning when reading in text files	`.split()`, `.join()`, `.replace()`, `.strip()` `.upper()`, `.lower()` - Case conversion	Process and manipulate text data, including biological	`.strip()` only removes from ends, not middle. `.split()` without

Concepts	Synergy	Syntax (Functions & Methods)	Objectives	Common Pitfalls / Notes
		`f"{}"` - String formatting	sequences like DNA/RNA strings	arguments splits on any whitespace.
Nested Structures	**Data organisation**: Complex data hierarchies **Matrix operations**: 2D data representation	`list[row][col]` - Multi-dimensional access List comprehensions for nested iteration	Work with complex data structures like matrices, and hierarchical data	Deep vs shallow copying can become critical in large, interweaving code bases
Data Type Conversion	**Data import**: Converting between formats **Type safety**: Ensuring correct data types	`list()`, `tuple()`, `set()` - Conversion functions `str()` - String conversion `sorted()` - Create sorted versions	Convert between different collection types whilst preserving or transforming data as needed	Converting to set removes duplicates and loses order, use it to quickly identify unique data points

Next ›