Strings

PF2

Learning Objectives

1. Understand string fundamentals
2. Apply **string operations** to manipulate and analyse sequence data
3. Utilise built-in **string methods** for searching, cleaning, and reformatting data
4. **Convert** strings to other data structures (lists and tuples)

Introduction

In the previous lesson, we have explored strings as both inputs and outputs. However, strings are much more than just output printed to your screen in a `print` statement; they are also a **data structure** in their own right, and can be manipulated and explored to extract information.

In this context, we can view them as a store of individual data points (each **character** pertaining to a single data point), stored as one long **sequence**: conceptually not too dissimilar to a list, in this respect. An obvious example for biologists of complex string data would be nucleic and amino acid sequences, where nucleotide or residue is a single point of information, as part of a longer sequence. In fact, many Python-based programs exist that can efficiently find and match specific string sequences across multiple DNA samples. In this section of the lesson, we will try to demonstrate our own approach to this.

Recap: creating a string

Any character, alphabetical, grammatical, numerical *etc.* can be stored as a string data-type *via* **single** (`'...'`) or **double** (`"..."`) **quotation marks** around the data, or by using the `str()` class (which typecasts data such as integers, floats and booleans into a string).

NOTE

It is **fine to use either single or double quotations** to create a string, although it is considered good practice to use and adhere to one in your Python script. However, there are a few occasions where you need to mix quotations, such as actively incorporating quotation marks inside your string, i.e. `'To quote Aristotle: "This is the proper way to use nested quotations"'`. If we were limited to only a single or double quote syntax, writing a sentence like this would result in the premature termination of the string.

```
alphabet_string = "abcdfeg"
numerical_string = str(12345) # can also be: numerical_string = "12345"
mixed_string = "abc-123"

print(alphabet_string)
print(numerical_string)
print(mixed_string)
```

```
abcdfeg
12345
abc-123
```

You can check if anything is a string by calling the `type()` class. If the data is a string, it should return `<class 'str'>`, when printed.

```
print(type(alphabet_string))
```

```
<class 'str'>
```

Numbers as strings

When numerical data is converted to a string, it will *lose the attributes associated with being an integer or float* data type. It is therefore important to check - when importing data - that your numerical data is not mistakenly being imported as a string; a common accident when data is parsed from one mode to another.

```
# No matter the characters within the string, they are all the same data type

type(alphabet_string) == type(numerical_string)
```

```
true
```

It is crucial to note that operations (such as mathematical operations) will perform differently on data that is a string, compared to the same sequence of numbers stored as a numerical data type:

```
numerical_true = 12345

numerical_string = str(numerical_true)

print("Multiplying an integer:                                        ",
      numerical_true, "---->", numerical_true * 2)

print("\nMultiplying the same numerical sequence, stored as a string:   ",
      numerical_string, "---->", numerical_string * 2)
```

```
Multiplying an integer:                                        12345 ---
-> 24690

Multiplying the same numerical sequence, stored as a string:    12345 ---
-> 1234512345
```

In the cell below, have a play with different operations on the string, we'll explore some in more detail further on. Note - many won't work and will throw an error, be sure to note the error as they are common ones.

```
practice_string = "L2D"
```

Run Code

```
Ready to run code!
```

String fundamentals

As you will have seen with data types, data structures also have fundamental aspects to them which standardises their usage. It is important to understand these aspects, so that you better understand how your data is stored and augmented throughout your Python script.

Immutability

Strings **cannot be changed after they are instantiated**. The data structure itself is **immutable**. If you were to augment the string through operations or methods, Python will instead create a new string to be stored as another variable if needed. This means you can always be assured your original data will be unchanged.

In the cell below, let's use the + to concatenate new characters into our string, using an F-string. Note how - inspite of the change - it is not overstored into the variable containing the original string, which is subsequently still recallable.

```
original_sequence = "ATCGATCG"
print(f"Concatenated sequence: {original_sequence + 'GGGGGGGG'}"), # Concatenate a
print(f"Original (unchanged):  {original_sequence}")
```

```
Concatenated sequence: ATCGATCGGGGGGGGG
Original (unchanged):  ATCGATCG
```

Case-sensitive

Python treats **uppercase** and **lowercase** as *different characters*. This means that extra dilligence should be afforded when importing data, to check that it was inputted, correctly. Many experimental pipelines have been held up by one user using lower case when another uses upper case.

We'll demonstrate some methods later that convert from one case to the other.

```
gene1 = "TP53"  # Upper case.
gene2 = "tp53"  # Lower case.

print(f"Are these genes equivalent? \n\n{gene1 == gene2}")
```

```
Are these genes equivalent?

False
```

Indexing

Much like a list or tuple, **strings can be indexed**, allowing the use of operations such as **slicing**. Each character is an **ordered item** in the string, exactly as each element is within a list. In fact, the same basic slicing operations that we previously covered in the lists section of this material, can be applied to strings.

```
rna_seq = "AUGGGCAGUCCGUAA"

print(f"Original sequence: {rna_seq}\n")

```

```
print(f"Slicing first nucleotide: {rna_seq[0]}")
print(f"Slicing last nucleotide: {rna_seq[-1]}")

print(f"Slicing RNA without start codon: {rna_seq[3:]}") # We sliced downstream of
```

```
Original sequence: AUGGGCAGUCCGUAA

Slicing first nucleotide: A
Slicing last nucleotide: A
Slicing RNA without start codon: GGCAGUCCGUAA
```

In th ecell below, isolate the termination codon from the string in variable `rna_seq`.

HINT: RNA stop codons can be either UAA, UAG or UGA.

```
rna_seq = "AUGGGCAGUCCGUAA"

```

Run Code

```
Ready to run code!
```

Comparable / sortable

Strings can also be **sorted** or **compared** based on their characters. Much as we can order various numbers by ascending or descending value, we can also sort strings based on the ranking of their characters.

Python compares strings lexicographically (in dictionary-like order) using **ASCII values**.

This means:

1. Numbers come before letters
2. Uppercase letters come before lowercase letters
3. Letters are ordered alphabetically

FACT

An **ASCII value** is a numerical code that represents a specific character in the *American Standard Code for Information Interchange* system, allowing computers to store and interpret text as numbers. For example, the character 'A' has an ASCII value of 65, 'a' is 97, and '0' is 48.

Observe how this logical behaves:

```
# Comparisons between strings is not always obvious

ex1 = "exp_10"
ex2 = "exp_2"

print(ex1 > ex2)
```

```
False
```

For strings, the comparison occurs from left to right in its search for differing characters. In the example above, the first 4 characters - `exp_` - are identical. The 5th are different with **1** coming before **2**.

If you want to ensure you sample names can be ordered properly, it is important to **zero-pad** you strings when entering them. That is - 'experiment 1' would be zero-padded to 'experiment 01'.

Zero-padding is demonstrated in the cell, below:

```
# If your samples run into the double digits, remember to add zeros in front of the

samples = ["exp_01", "exp_10", "exp_02", "exp_21"]

print(sorted(samples))
```

```
['exp_01', 'exp_02', 'exp_10', 'exp_21']
```

String operations

Python also supports several basic operations with strings, similar to those found with lists:

Concatenation

A new character, or sequence of new characters, can be added (or concatenated) onto the end of another string, increasing its size and information. This is done using the logical operator: `+`.

The cell below demonstrates how `+` behaves differently with a list, compared with a string.

```
start_list = ['1']
start_string = '1'

print("Adding to a list:", start_list + ['1'])
print("Adding to a string:", start_string + '1')
```

```
Adding to a list: ['1', '1']
Adding to a string: 11
```

In many ways both the string and list above are similar: they are both a sequential store of the character '1'.

Adding strings together can is also useful as a quick method to create unique ID's given two pieces of existing data. For example:

```
study = "CARDIO"
patient_num = "001"
patient_id = study + "_" + patient_num
print(patient_id)
```

```
CARDIO_001
```

Multiplication

Multiplying a string **repeats the whole string by the given number**. There are not many uses for this within biology, but it does make for a nice visual break when printing your output.

```
title = " BACTERIAL GROWTH ASSAY "
border = "=" * len(title)
print(border)
print(title)
print(border)
```

```
========================
 BACTERIAL GROWTH ASSAY
========================
```

```
# Have a play around with a string using the above operations

study = "test_string"

```

Run Code

```
Ready to run code!
```

`in` operation

Often the information we are looking for is hidden within a larger string. Using the `in` operation we can determine if a unique sub-string is within a larger string. The `in` operation won't isolate the string you are looking for, but often all you need to know is if one data point is present for an action to be taken.

```
# protein sequence
protein_sequence = "MKWVTFISLLFLFSSAYSRGVFRRDAHKSEVAHRFKDLGEENFKALVLIAFAQYLQQ"

# Check for signal peptides and domains
signal_peptide = "MKW"
hydrophobic_region = "FFF"
binding_motif = "KSEVAHRFK"

print(f"Has signal peptide:             {signal_peptide in protein_sequence}")
print(f"Contains hydrophobic region:    {hydrophobic_region in protein_sequence}")
print(f"Contains binding motif:         {binding_motif in protein_sequence}")
```

```
Has signal peptide:             True
Contains hydrophobic region:    False
Contains binding motif:         True
```

```
# Have a play around finding sub-sequences in `protein_sequence`
```

Run Code

```
Ready to run code!
```

PRACTICE EXERCISE

1.

You have a tuple of strings containing the approved treatments for a clinical trial, `approved_drugs`:

1. You've noticed "insulin" is misspelled, it's missing the last two letters. Add the missing letters back to it.

```
approved_drugs = ("metformin", "insul", "glipizide", "pioglitazone")
```

Run Code

```
Ready to run code!
```

String methods

The string data type also comes with **built-in methods** that quickly and neatly perform common tasks with letter based data, such as *finding sub-strings*, *removing unwanted characters* (such as whitespace), and *reformatting to a uniformed state*. We will go through a few of the common string methods, and provide a table that contains few more.

There are many more string methods than demonstrated or mentioned here, some of which fit very specific circumstances. They can all be found in the Python documentation here.

`.find()`

Rather than using the `in` to confirm the presence of sub-strings we can use the `find()` method. This method will **search for the given string and return the index of the first character**, if not present it will return **-1**. The index can then be used to extract the subsequent string.

```
medication = "ibuprofen_200mg_tablet"

# Find the position of substring (returns -1 if not found)
print(f"Index of '200gm': {medication.find('200mg')}\n")
print(f"Index when not found: {medication.find('400gm')}")    # -1 (not found)
```

```
Index of '200gm': 10

Index when not found: -1
```

```python
# Isolate the rest of the string using the returned index
index_start = medication.find("200mg")

print(f"Isolated string: {medication[index_start:]}") # Prints all characters downs
```

```
Isolated string: 200mg_tablet
```

Count

The `count()` method allows the user to find the number of times a particular character or sequence appears within a string, as demonstrated, below:

```python
dna_sequence = "ATCGGATCGGACGG"

# Count occurrences
print(f"Number of instances of 'AT': {dna_sequence.count("AT")}")
print(f"Number of instances of 'CG': {dna_sequence.count("GG")}")
```

```
Error: Traceback (most recent call last):
  File "/lib/python311.zip/_pyodide/_base.py", line 571, in
eval_code_async
    await CodeRunner(
          ^^^^^^^^^^^
  File "/lib/python311.zip/_pyodide/_base.py", line 268, in __init__
    self.ast = next(self._gen)
               ^^^^^^^^^^^^^^^
  File "/lib/python311.zip/_pyodide/_base.py", line 145, in
_parse_and_compile_gen
    mod = compile(source, filename, mode, flags | ast.PyCF_ONLY_AST)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "<exec>", line 4
    print(f"Number of instances of 'AT': {dna_sequence.count("AT")}")
                                                                  ^^
SyntaxError: f-string: unmatched '('
```

Other methods for searching strings

Method	Description	Example
`find(sub)`	Find substring (returns index or -1)	`"hello".find("ll") → 2`
`rfind(sub)`	Find substring from right	`"hello".rfind("l") → 3`
`index(sub)`	Find substring (raises error if not found)	`"hello".index("ll") → 2`
`rindex(sub)`	Find substring from right (raises error)	`"hello".rindex("l") → 3`
`count(sub)`	Count occurrences	`"hello".count("l") → 2`
`startswith(prefix)`	Check if starts with	`"hello".startswith("he") → True`
`endswith(suffix)`	Check if ends with	`"hello".endswith("lo") → True`

Case manipulation

As we learned earlier, Python **strings are case-sensitive**. This can have considerable effects on your analysis, as groups that are meant to be the same may have small capitalisation differences. If you are unable to prevent these happening at the source, you can augment them using the `.lower()` and `.upper()` string methods.

```
# Going back to our earlier example

gene1 = "TP53"
gene2 = "tp53"

print(f"Are these genes equivalent? {gene1 == gene2}")

# we can change either string to be in upper or lower case

print(f"Are these genes equivalent using .lower()? {gene1.lower() == gene2}")

print(f"Are these genes equivalent using .upper()? {gene1 == gene2.upper()}")

```

```
Are these genes equivalent? False
Are these genes equivalent using .lower()? True
Are these genes equivalent using .upper()? True
```

Other case methods

Method	Description	Example
`upper()`	Convert to uppercase	`"hello".upper() → "HELLO"`
`lower()`	Convert to lowercase	`"HELLO".lower() → "hello"`
`capitalize()`	Capitalise first letter only	`"hello world".capitalize() → "Hello world"`

Method	Description	Example
`title()`	Capitalise each word	`"hello world".title() → "Hello World"`
`swapcase()`	Swap upper/lower case	`"Hello".swapcase() → "hELLO"`
`casefold()`	Aggressive lowercase for comparisons	`"ß".casefold() → "ss"`

Cleaning

Another common issue when dealing with strings is **unwanted characters**, such as *whitespace* or *special characters*, that are an *artefact* of, say, the machine recording the experiment, or of parsing the content into Python. This is especially common in *FASTA files*, which often contain many millions of characters, or even billions, of characters.

The `.strip()` string method, for example, can be used to remove **whitespace** and *escape sequences** (such as `\n`, which returns a newline character).

```
# Remove whitespace and escape characters (common when reading files)
messy_gene = "  EGFR  \n"
clean_gene = messy_gene.strip()
print(clean_gene)
```

```
EGFR
```

Another method - `.replace()` - can be useful for finding a certain character, and replacing it with another. The first argument it accepts is the sequence being searched, and the second argument, is the sequence intended to replace it.

```
# Replace characters
sequence = "ATCG-NATCG"
clean_sequence = sequence.replace("-", "").replace("N", "")
print(clean_sequence)
```

```
ATCGATCG
```

The `.split()` method is also particularly useful when dealing with FASTA files that have varying seperators.

In the example below, we split the sequence header on the `|` character, assigning each split section of the string to the element of a new list. We can then use indexing to retrieve just the accession number, for instance, from this list, allowing us to swiftly retrieve just the information we are looking for, from a string, using minimal code.

```
```

```python
# Parsing a FASTA header
fasta_header = ">gi|123456|ref|NM_000546.5| tumor protein p53 (TP53), mRNA"

# Remove the '>' and split by '|'
header_parts = fasta_header[1:].split('|')
print(f"FASTA components: {header_parts, type(header_parts)}\n")

# Extract specific information
accession = header_parts[3]
print(f"Accession number: {accession}")
```

```
FASTA components: (['gi', '123456', 'ref', 'NM_000546.5', ' tumor protein p53 (TP53), mRNA'], <class 'list'>)

Accession number: NM_000546.5
```

Other modification methods

Method	Description	Example
`replace(old, new)`	Replace occurrences	`"hello".replace("l", "x")` → `"hexxo"`
`strip()`	Remove whitespace from ends	`" hello ".strip()` → `"hello"`
`lstrip()`	Remove whitespace from left	`" hello ".lstrip()` → `"hello "`
`rstrip()`	Remove whitespace from right	`" hello ".rstrip()` → `" hello"`
`removeprefix(prefix)`	Remove prefix if present	`"hello".removeprefix("he")` → `"llo"`
`removesuffix(suffix)`	Remove suffix if present	`"hello".removesuffix("lo")` → `"hel"`

PRACTICE EXERCISE

2.

You're cleaning up a messy dataset of gene names and sample IDs from different laboratories. Each lab has used different formatting conventions that need to be standardised. Using the provided data:

1. Remove any leading/trailing whitespace from the gene name
2. Convert the gene name to uppercase for consistency
3. Replace any hyphens (-) with underscores (_) in the gene names
4. For the sample ID, split it at the underscore and create a formatted ID using f-strings: f"LAB-{lab_code}_SAMPLE-{number}"
5. Create a final summary string that combines the cleaned gene name and formatted sample ID

```
# Messy data
gene_name = "  brca-1  "
sample_id = "gilestro_024"
```

Run Code

```
Ready to run code!
```

Converting strings into other data structures

Lists

Converting a string to a **list** can be extremely useful when you need to manipulate individual characters or when you want to analyse the frequency of specific elements. The `split()` method is particularly powerful for parsing structured data, whilst the `list()` class converts each character into a separate list element.

```
# Convert each character to a list element
dna_sequence = "ATCGATCG"
```

```
nucleotides = list(dna_sequence)
print(f"Individual nucleotides: {nucleotides}\n")

# Split a string by delimiter (useful for CSV data)
sample_data = "sample1,control,25.6,positive"
data_fields = sample_data.split(",")
print(f"Data fields: {data_fields}")
```

```
Individual nucleotides: ['A', 'T', 'C', 'G', 'A', 'T', 'C', 'G']

Data fields: ['sample1', 'control', '25.6', 'positive']
```

PRACTICE EXERCISE

3.

Convert the following amino acid sequence into a list and count how many hydrophobic amino acids (A, I, L, M, F, W, Y, V) it contains.

```
protein_sequence = "MKWVTFISLLFLFSSAYS"
hydrophobic = ['A', 'I', 'L', 'M', 'F', 'W', 'Y', 'V']

# Your code here

```

Run Code

```
Ready to run code!
```

Useful `string`-to-`list` methods

Method	Description	Example
`list(string)`	Convert each character to list element	`list("ATCG")` → `['A', 'T', 'C', 'G']`
`split()`	Split by whitespace	`"hello world".split()` → `['hello', 'world']`
`split(delimiter)`	Split by specific character	`"A,T,C,G".split(",")` → `['A', 'T', 'C', 'G']`

Method	Description	Example
`splitlines()`	Split by line breaks	`"line1\nline2".splitlines() → ['line1', 'line2']`
`partition(sep)`	Split into 3 parts at first separator	`"a=b=c".partition("=") → ('a', '=', 'b=c')`
`rpartition(sep)`	Split into 3 parts at last separator	`"a=b=c".rpartition("=") → ('a=b', '=', 'c')`

Tuples

Converting strings to tuples follows similar principles to lists, but creates an **immutable sequence**. This can be particularly *useful when you need to ensure your sequence data remains unchanged* throughout your analysis, or when you want to use the sequence as a dictionary key (we will discuss dictionaries in Python Fundamentals 3).

```python
# Convert string to tuple of characters
codon = "ATG"
codon_tuple = tuple(codon)
print(f"Codon as tuple: {codon_tuple}")
print(f"Data type: {type(codon_tuple)}")

# Useful for creating immutable sequence identifiers
sample_id = "EXP_001_CTRL"
id_parts = tuple(sample_id.split("_"))
print(f"Sample ID parts: {id_parts}")
```

```
Codon as tuple: ('A', 'T', 'G')
Data type: <class 'tuple'>
Sample ID parts: ('EXP', '001', 'CTRL')
```

PRACTICE EXERCISE

4.

Create a tuple from the following restriction enzyme recognition sequence and use it to check if it's *palindromic*.

1. Covert the string to a list
2. Save the reversed list to a new variable
3. Check their equivalence

```
restriction_site = "GAATTC"

# Your code here

```

Run Code

```
Ready to run code!
```

Key differences: lists *vs.* tuples from strings

Aspect	List	Tuple
Mutability	Mutable (can be changed)	Immutable (cannot be changed)
Use Case	When you need to modify sequence	When sequence should remain constant
Dictionary Key	Cannot be used as dict key	Can be used as dict key
Memory	Uses more memory	Uses less memory
Performance	Slightly slower access	Faster access
Methods	Many methods (append, remove, etc.)	Few methods (count, index)

Summary In this section, we explored **strings — ordered sequences of characters used to store and manipulate text in Python. We learned that strings can be created using **single** (`'...'`) or **double** (`"..."`) quotation marks, and that any type of character — alphabetical, grammatical, numerical *etc.* — can be represented as a string. We also saw how the `str()` class can convert other data types into their string form. The lesson introduced **escape sequences** (such as `\n` for a new line), **f-strings** for formatted output, and the concept of **ASCII values**, which map characters to numerical codes (e.g. `'A'` is `65`). Together, these tools allow us to store, display, and manipulate textual data effectively in Python.**

‹ Previous

Next ›