Tuples

PF2

Learning Objectives

1. Understand the concept of immutability in Python
2. Create and work with tuples
3. Apply tuple packing and unpacking techniques
4. Understand when to use tuples vs lists

Introduction

We have just learnt about Python lists, a general data store that is modifiable or mutable. However, there are occasions where we *don't want our data to be changed*, so that it remains constant and consistent: in other words we want it to be **immutable**.

For such applications, we can use a **tuple**, a data structure whose contents *cannot be modified* in any way, once the strucutre is created or instantiated. Such an example might be experimental parameters for a run. These are fixed in the past, would not need modifying, and are likely to be used repeatably throughout analysis.

To read about tuples in more depth,documentation of `tuples` from Python.org, is linked.

Creating a tuple

The most common way to implement a tuple in Python is to place comma-separated values inside **round parentheses**:

```
x = (1, 2, 3)
```

As seen with lists and square brackets `[]`.

You can also create a tuple *without* parentheses:

```
x = 1, 2, 3
```

In this case, Python acknowledges that a tuple is *implied* through the comma separation, creating the tuple through an operation known as **packing**. The former is referred to as an **explicit tuple**, and the latter, an **implicit tuple**.

NOTE

When creating a `tuple`, it is considered best practice to do so **explicitly**, using round parentheses `()`.

HISTORY

There are numerous ways of doing things in Python, all with their own potential benefits to different users and scenarios. However, there are good practices to coding in Python. One guide (or more of a philosophy) to how you set out your code is called the **Zen of Python**, a collection of 19 guiding principles for writing computer programs, written by Tim Peters, a major contributor to the creation of Python itself.

One line stands out for beginners:

```
Explicit is better than implicit.
```

We should always be making our code as readable as possible for others and ourselves, which means taking the time to use code that explicitly shows what is happening. For example, this principle encourages using round parentheses `()` when creating a tuple, even when they're not strictly required.

KEY TERMS

Explicit: Clearly stated or defined code that shows exactly what is happening, following the Zen of Python principle "Explicit is better than implicit". For example, using parentheses `()` when creating tuples even when they're not strictly required.

Implicit: Code behaviour that is inferred or understood without being directly stated, such as creating tuples without parentheses using comma separation.

```
1 # The Wnt pathway coreceptors will not be changing, so we save them as a tuple
2
```

```
coreceptors = ('Frizzled', 'LRP')

print(type(coreceptors))
```

```
<class 'tuple'>
```

```
# Printing them retains the () brackets

print(coreceptors)
```

```
('Frizzled', 'LRP')
```

Accessing values inside a tuple works the same as accessing a list, using square brackets `[]`.

```
print(coreceptors[0])
```

```
Frizzled
```

However, different from a list, a tuple is **immutable**, so we cannot reassign or modify the values within

```
# Run this cell to return a TypeError due to trying to reassign one of the values

coreceptors[0] = 'new'
```

Run Code

```
Ready to run code!
```

Accessing a Tuple

The principles of indexing and slicing a tuple are identical to those of a list. You can access elements using positive and negative indexing, and create slices using the same `[start:end]` notation.

```
# DNA bases tuple
dna_bases = ('A', 'T', 'G', 'C')

print('First base:', dna_bases[0])     # Positive indexing
print('Last base:', dna_bases[-1])     # Negative indexing
```

```
print('First two:', dna_bases[:2])     # Slicing
```

```
First base: A
Last base: C
First two: ('A', 'T')
```

```
# Practice indexing with the tuple below
amino_acids = ('Ala', 'Gly', 'Val', 'Leu', 'Ile', 'Pro')

# Access the third amino acid
# Access the last two amino acids using slicing
```

Run Code

```
Ready to run code!
```

PRACTICE EXERCISE

1.

Create a tuple called `genetic_code` containing the four DNA bases: 'A', 'T', 'G', 'C'. Then:

- Print the length of the tuple
- Access and print the second base (index 1)
- Print the last base using negative indexing

```
# Create your genetic_code tuple here
```

Run Code

```
Ready to run code!
```

Empty or single tuples

A `tuple` can be empty or only have a single value (singleton) stored within it.

The use of empty tuples is not immediately obvious, but they do have their use case in maintaining data type consistency, on occasions where the entry has no data, for instance. Remember: a tuple cannot be modified after it is instantiated, and so it will stay empty, unless it is overwritten or replaced.

```
empty_tuple = tuple() # empty_tuple = () also works

print(f"Tuple: {empty_tuple}, Type: {type(empty_tuple)}, Length: {len(empty_tuple)}
```

```
Tuple: (), Type: <class 'tuple'>, Length: 0
```

Singleton tuples are needed frequently, but have a creation quirk that is worth addressing. *You cannot create a singleton tuple with just round parentheses around an item* `(item)`. Instead, you **must** provide a *trailing comma after the item*: `(item,)`. Without the comma, Python will interpret the parentheses as a grouping and not a tuple, which can cause a lot of common bugs.

```
# Without the comma an error will be given
# Try this for yourself by removing the comma, and rerunning the cell

singleton = (1,)

print(f"Tuple: {singleton}, Type: {type(singleton)}, Length: {len(singleton)}")
```

Run Code

```
Ready to run code!
```

Conversion to a tuple

Similar to list, we can convert other sequences to tuple using the `tuple()` class.

NOTE

Just like `list()`, `tuple()` is technically a **class** and *not* a function. And yet, the way that we use it - it behaves like a function. Essentially, we pass it data, and use it to return an object containing data that is **typecasted** (where its type is forced) into that of type `tuple`.

In the code cell below, let's create a list, and check its type.

```
# Our behaviour list is now complete with no further editing

behaviours = ['foraging', 'grooming', 'resting']

print(type(behaviours))
```

```
<class 'list'>
```

And now lets create a new tuple object, and typecast the items in the list into the elements of a new tuple, using the `tuple()` class.

```
# So we make it a tuple to prevent any accidental modification

behaviours_tuple = tuple(behaviours)

print(behaviours_tuple)
print(type(behaviours_tuple))
```

```
('foraging', 'grooming', 'resting')
<class 'tuple'>
```

Experiment for yourself in the code cell below, typecasting the list into a tuple object.

```
# Convert a list of enzyme names to a tuple
enzyme_list = ['amylase', 'pepsin', 'trypsin', 'lipase']

# Convert to tuple and print both the result and its type
```

Run Code

```
Ready to run code!
```

Item Mutability

Whilst a tuple itself is *immutable*, the mutability of its items depends on the data type of that value. It can therefore contain both mutable and immutable objects. For the sake of explicitness, this basically means that a tuple *can* contain mutable objects but is, itself, *immutable*.

In the code cell below, we will create a tuple that contains a list (a mutable data structure) as one of its items:

```
# This tuple's items are: (immutable, immutable, immutable, immutable, MUTABLE)
mixed_tuple = (1, 2.5, 'abc', (3, 4), [5, 6])

print(mixed_tuple)
```

```
(1, 2.5, 'abc', (3, 4), [5, 6])
```

Mutable objects inside a tuple may still be changed. Just because we are storing a mutable list inside a tuple, does not mean that we cannot edit the contents of that list. It is only the outer structure of the tuple itself that cannot be changed.

Let's slice into that list in this code cell, and demonstrate how we can change its first element using adjacent pairs of square brackets, and reassigning the first list item (index 0), by replacing it with a string.

For the sake of explicitness, the first line of code in the cell below accesses the last item in the tuple using negative indexing, and in the second square bracket pair, accesses the first item in the sublist, and reassigns its value from '5' to 'index modified'.

```
# Accessing the items inside the mutable object allows us to modify them

mixed_tuple[-1][0] = 'index modified'

print(mixed_tuple)
```

```
(1, 2.5, 'abc', (3, 4), ['index modified', 6])
```

```
# Applying a method that acts in-situ to the item will also modify it

mixed_tuple[-1].append('append modified')

```

```
print(mixed_tuple)
```

```
(1, 2.5, 'abc', (3, 4), ['index modified', 6, 'append modified'])
```

```
# We cannot remove the list from the tuple,
# but we can empty it by clearing its items:

mixed_tuple[4].clear()

print(mixed_tuple)
```

```
(1, 2.5, 'abc', (3, 4), [])
```

★ **FACT**

Why *can* we change mutable objects inside a tuple?

Members of a tuple are *not directly stored* in memory. An immutable value (e.g. an integer) has an existing, predefined **reference** in memory. When used in a tuple, it is this reference that is *associated* with the tuple, not the value itself.

A mutable object, however, *lacks a predefined reference* in memory, and is instead created on request somewhere in your computer's memory. While we can never change a predefined reference, we can manipulate something we have defined ourselves. When we make such an alteration, the location of our mutable object in memory may change, but its reference, which is what is stored in the tuple, remains identical.

PRACTICE EXERCISE

2.

You're working with experimental conditions that should never change during analysis. Create the following tuples:

1. An empty tuple called `placeholder_results`
2. A singleton tuple called `control_group` containing only the string 'untreated'
3. Try to modify the second element of this tuple: `('constant', 'fixed', 'immutable')` to see the error

```
# Create your tuples and test immutability here
```

Run Code

```
Ready to run code!
```

Packing and unpacking

As previously discussed, a tuple may also be constructed without parentheses. This is an implicit operation in Python, and is known as **packing**.

KEY TERMS

Packing: The process of combining multiple values into a single tuple, either explicitly with parentheses `(a, b, c)` or implicitly with comma separation `a, b, c`. Python automatically interprets comma-separated values as a tuple.

Unpacking: The process of extracting values from a tuple and assigning them to individual variables, making code more readable and allowing work with individual elements.

For example, `x, y, z = coordinates` extracts three values from a coordinates tuple, and stores them in the order they appear.

Let's demonstrate packing in this cell, creating a new tuple called `my_tuple`, in the process:

```
my_tuple = 1, 2, 3 # Implicitly packing 3 numbers into a tuple (not using round bra

# my_tuple = (1, 2, 3) <- This would produce the same result, and is just explicitl

print(f"My tuple contains: {my_tuple} and is of type {type(my_tuple)}.")
```

Run Code

```
Ready to run code!
```

And in this cell, let's *unpack* that tuple's items into three variables called a, b and c.

This is also an example of a **multiple assignment**, where we are using a single line of code and a single = operator to assign values to three variables, at once. (The alternative would be to assign each of them, individually, on separate lines).

```
a, b, c = my_tuple # Creates three variables, and storing each of the tuple's items

print(f"The contents of a = {a}", f"The contents of b = {b}", f"The contents of c =
```

Run Code

```
Ready to run code!
```

⚠ WARNING

Implicit processes should be used sparingly. As always, the **more coherent and explicit the code, the better it is**. However, packing and unpacking are widely used Python idioms that make code more readable in certain contexts.

For clarity, here's a second example, where we *explicitly* create the tuple using round brackets. We consider this to be better coding practice, but both packing techniques are considered valid.

```
dimensions = (1, 2, 3, 5, 7, 11)

print(dimensions,          # Our new tuple.
      type(dimensions),    # Checks its type.
      len(dimensions),     # Checks its length (its total number of items).
      sep='\n')
```

```
(1, 2, 3, 5, 7, 11)
<class 'tuple'>
6
```

As we can see, our tuple contains 6 elements (or items). Let's try to unpack the first 3, and see what happens:

```
1 x, y, z = dimensions
2
3 print("First 3 dimensions:", x, y, z)
```

```
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 394, in run_async
    coroutine = eval(self.code, globals, locals)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "<exec>", line 1, in <module>
ValueError: too many values to unpack (expected 3)
```

Python immediately throws a ValueError. When unpacking a `tuple`, the **number of variables on the left must match the number of elements in the tuple**.

NOTE

Why do we need to know this?

Packing and **unpacking** are two processes employed when you want to store the returned values from a function (if there are multiple values). If all returned values are stored in one variable, they will be stored as a tuple. Unpacking allows you to separate them into individual, meaningfully named variables. This will become clearer when we discuss functions at greater length in our Python Fundamentals 4 lesson.

PRACTICE EXERCISE

3.

You're analysing protein structure data. Each protein record contains: name, molecular_weight, length, and function.

1. Create a tuple called `insulin_data` with the values: ('Insulin', 5808, 51, 'hormone')
2. Use unpacking to extract these values into separate variables with meaningful names
3. Print a formatted message: "Insulin is a hormone with molecular weight 5808 and length 51"
4. Create another tuple using packing (without parentheses) for haemoglobin: ('Haemoglobin', 64500, 574, 'oxygen transport')

```
# Complete the protein analysis tasks here
```

Run Code

```
Ready to run code!
```

Advanced Unpacking Techniques

Python also supports more advanced unpacking patterns, including the *operator. The* will collect any remaining items into a list, allowing you to unpack a variable number of elements from a sequence. This is particularly useful when you know some specific positions but want to capture the rest as a group.

```
# Unpacking with the * operator
gene_expression_data = ('BRCA1', 2.3, 4.1, 3.7, 5.2, 1.8)

gene_name, *expression_levels = gene_expression_data

print('Gene:', gene_name)
print('Expression levels:', expression_levels)
```

```
print('Type of expression levels:', type(expression_levels))
```

```
Gene: BRCA1
Expression levels: [2.3, 4.1, 3.7, 5.2, 1.8]
Type of expression levels: <class 'list'>
```

The `*` operator captures all remaining elements as a list (not a tuple), making it easy to work with variable-length data.

FACT

Named Tuples: There is another type of tuple in Python called a `namedtuple`. This allows the members of a tuple to be named independently (e.g. `member.name` or `member.age`), eliminating the need for unpacking in some cases.

It is not a built-in tool but it is included in the standard library, so you won't have to install it. It was originally implemented by Raymond Hettinger, one of Python's core developers. While not covered in detail here, it's worth knowing about for more advanced applications.

Summary

In this section, we learned about `tuple`, another type of data structure within Python, and one which is **immutable**.

Tuples are particularly useful in biological programming for storing data that should *remain constant* throughout analysis, such as experimental conditions, genetic codes or protein properties. Their immutability provides *data integrity* whilst their ability to be unpacked makes them excellent for returning multiple values from functions.

Understanding when to use tuples versus lists is crucial: use tuples for data that should not change, and lists for collections that need to be modified during program execution.