Python Fundamentals

Tuples, sets, dictionaries

Objectives:

- Container Data Types
- Tuple
- Immutable vs Mutable
- Sets
- in and not in operators
- Dictionaries

Container Data Types

- By now you would be familiar with the **list** built-in data type. It provides a way to store a collection of (usually homogenous) values. A collection of numbers, for example.

- The **list** built-in data type is considered a **sequence** type, because it stores values in a specific sequence.

- **Sequence** data types belong to a larger category, called **container** data types.

Container Data Types

- A **container** is a data type that *contains* a number of values. Some examples of other **container** data types include:

- Tuples
- Sets
- Dictionaries

Tuple

- A tuple is a sequence, much like a list. A tuple differs from a list in two important ways.

- It is **immutable**, meaning it cannot be changed after initialization.

- It is usually intended for storing **heterogeneous** data. That is, different types of data. As opposed to **homogenous** data, which is the same type.

Tuple

Declaring a tuple is quite simple; literal values separated by a comma. All values can be contained by parenthesis.

```
# A tuple of heterogeneous values
t = 12345, 54321, 'hello!'

# A tuple of homogeneous values
u = (1, 2, 3, 4, 5)

# A tuple created with the constructor
v = tuple('abc')
# Produces: ('a', 'b', 'c')
```

Tuple

Once a tuple is initialized, we CANNOT modify its contents. This includes appending or overwriting elements.

```
# Initialize a tuple
t = 12345, 54321, 'hello!'

print(t[0])
# Prints: 12345

t[0] = 88888
# Produces the following error:
Traceback (most recent call last):
  File "container_slide_samples.py", line 19, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuple Packing and Unpacking

- When we take a number of literal values, separated by commas, and assign it to a variable, this is known as *tuple packing*.

- We can also take each element in a tuple and assign it to a corresponding variable. This process is known as *sequence unpacking*.

- It's here that tuples can become very useful.

Tuple Packing and Unpacking

Below are examples of performing tuple packing and sequence unpacking.

```
# Initializing a tuple uses tuple packing
t = 12345, 54321, 'hello!'

# Using sequence unpacking
# We must have a variable for each element
x, y, z = t

# You can unpack any sequence type
l = [2, 4, 6] # Initialize a list
a, b, c = l # Unpack the elements
```

Immutable vs Mutable

- As mentioned earlier, tuples in Python are considered **immutable**. This is the opposite of **mutable**.

- The word is derived from the original Latin *mutare*, meaning ‘To change’.

- In other words, whether a data type can be modified after initialisation.

Immutable vs Mutable

- Whether a data type is **immutable** or **mutable** is a core characteristic of many programming languages.

- Some languages, for example, **only** have immutable data types. This is common amongst Functional Programming languages.

- Python has a mix of both.

Immutable vs Mutable

- Some examples of **immutable** data types include tuples, strings, integers, floats, and Booleans.

- Some examples of **mutable** data types include lists, sets, dictionaries, and custom classes.

Sets

- As has been mentioned, sets are another of the container data types. They are also mutable.
- They differ from lists in the sense that every element in a list must be a unique value.
- All elements within a set are unordered.

Sets

A set is initialized using comma separated values, surrounded by curly braces.

```
# Initializing a set
basket = {'apple', 'orange', 'apple', 'pear', 'banana'}

# All duplicates are automatically removed
print(basket)
# Prints: {'pear', 'apple', 'banana', 'orange'}

# Initializing an empty set
empty = set()
```

Sets

You can also add and remove elements of a set. If the element already exists, the existing element is overwritten.

```
# Initializing an empty set
basket = set()

# Adding an element
basket.add('orange')
# Adding the same element.
basket.add('orange')
# Contains: {'orange'}

# Removing an element
basket.remove('orange')
```

Sets

- Sets are especially useful for doing *set-based operations*. These include intersection, union, and difference.

- As well as methods, these operations can be performed with the relevant **operators**.

```
& == intersection
| == union
- == difference
^ == symmetric_difference
```

Sets

Below are some examples of using set-based operations, either with the operators or methods.

```
a = set('abracadabra') # Contains: a, r, b, c, d
b = set('alacazam') # Contains: a, l, c, z, m

# Intersect
a & b # Result contains: a, c

# Union
a.union(b) # Result contains: z, c, a, l, r, b, m, d

# Difference
a - b # Result contains: b, r, d
```

in and not in operators

- One operation that sets are very good at is *membership testing*.
- This involves determining whether a value exists within a container of values. A list, tuple, set, or dictionary.
- Membership testing is done using the **`in`** and **`not in`** operators.

in and not in operators

Below demonstrates the usage of the operators. Note that they can be used with any container type.

```
basket = {'apple', 'orange', 'apple', 'pear', 'banana'}
# Contains: pear, apple, banana, orange

'orange' in basket
# Returns: True

'crabgrass' in basket
# Returns: False
'crabgrass' not in basket
# Returns: True
```

Dictionaries

- As well as sequences (lists, tuples), and sets, Python also has **mapping types**.
- Mapping types *map* a *key* to a *value*. The key must be an **immutable** data type, and the value can be any type.
- Python currently has a single built-in mapping type, called a **dictionary**.

Dictionary

Dictionaries are initialized with key/value pairs. Each pair is separated by a colon, pairs are separated by commas, and all pairs are contained between a single pair of curly braces.

```
# Initialising some dictionaries
numbers = {'one': 1, 'two': 2, 'three': 3}
tel = {'jack': 4098, 'sape': 4139}

# Using the constructor
numbers = dict(one=1, two=2, three=3)

# Initialising an empty dictionary
tel = dict()
```

Dictionary

- Every *value* in a dictionary can be retrieved using its associated *key*.

- This is similar in principle to a list, where every *element* can be retrieved using its associated *index*.

- Unlike a list, a *key* doesn't have to be a number. It can be a string, integer, float, and more; so long as the data type is **immutable**.

Dictionary

Below we are initialising a new empty dictionary. We can then add, retrieve, or remove values using their associated key.

```
tel = dict()

# Adding a key/value pair
tel['irv'] = 4127

# Retrieving a value using a key
tel['irv'] # Returns: 4127

# Removing a key/value pair
del tel['irv']
```

Access Dictionary's keys and values

Below we are initialising a new empty dictionary. We can then add, retrieve, or remove values using their associated key.

```
# Prints all the keys
for name in tel:
    print(name)

# Finding if a name is in the keys
if name in tel:
    print(name, "is in the dictionary")

# Comparing value in a key:value pair
if num == tel[name]:
    print("correct number for", name)
```

Access Dictionary's keys and values

Below we are initialising a new empty dictionary. We can then add, retrieve, or remove values using their associated key.

```
# Prints key:value pairs: ("irv", "4127")
for item in tel.items():
    print(item)

# Unpacking a key:value pair
    name, num = item

# Prints all keys and values
print(tel.keys(), tel.values())
```

Types of Structured Data

Name	Description	Mutable or not	Example
String	A group of characters	Immutable (unchangeable)	fruit="apple" print(fruit[0]) # 'a'
List	A group of elements of any data type, accessible via their position in the list	Mutable (changeable)	result=["HD",70,100] print(result[1]) # '70'
Tuple	A group of elements of any data type, accessible via their position in the list	Immutable	cities=("Sydney","Melbourne") print(cities[1]) # 'Melbourne'
Set	A group of values, none of which are repeated, in no particular order	Mutable	fruit={"apple","orange"} print("orange" in fruit) # True
Dictionary	A group of key:value pairs, accessible via their key name	Mutable	age={"Bob":32,"Sue":25} print(age["Sue"]) # 25

References

Container type tutorial

https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

in and not in reference

https://docs.python.org/3/reference/expressions.html#in

Demonstration:

- Container Data Types
- Tuple
- Immutable vs Mutable
- Sets
- in and not in operators
- Dictionaries