Fundamentals of Computer Programming

Lecture slides - Lists and Tuples

Intro

- This lesson covers
  - Revisiting the **List** data-type
  - An overview of the list *methods*
  - List Comprehensions
  - Introducing **Tuples**
  - Creating and accessing Tuples
  - Tuples and function arguments
  - Lists vs Tuples

Intro

- We have already been introduced the **list** data-type in a previous lesson
- Just to recap, these allow a number of values to be associated together in an ordered fashion, e.g.

```
squares = [4, 9, 16, 25]
```

- Lists (like *Strings*) are a type of **sequence**, and thus support common features such as *indexing*, *slicing* and *concatenation*
- Lists can be *iterated* over using `for` statements
- Unlike strings, lists are **mutable**, that is - once created the elements can be added or removed, by indexing, slicing or calling *methods* such as `append()`

What is a method?

- The term **method** is related to a programming paradigm called Object Orientation (OO)
- We will look at OO specific support within Python later in the module
- A **method** is similar to a *function*, but associated with a specific data-type
  ↳ Difference is that it is associated to a specific data type
- Therefore *methods* available on one data-type, are not necessarily available on others
- Unlike a function, a method is called in the *context* of an object, this basically means the call is preceded by a variable name and a period, e.g.

```
squares.append(36)  # call append 'on' the squares list
```

A method is always associated with a particular function

List specific methods

- Like many types (*built-in* or otherwise) lists have a number of associated methods, we have already seen `append()` however there are also others -

  `extend()`, `insert()`

  `remove()`, `pop()`,`clear()`

  `sort()`, `reverse()`

  `index()`,`count()`, and `copy()`

- Although not shown here, these typically take additional *parameters*.
- Most of these methods, with the exception of `index()`,`count()` and `copy()`, **mutate** the list

List method examples

- Let's examine a couple of these methods in more detail -
- The `extend()` method allows us to append multiple values, by providing an *iterable* value as the parameter, e.g.

```
squares.extend([49, 64, 100])
```

Adds 3 more elements to a list.

- The `insert()` method allows us to insert a single *value* at a specific *index* position, (both *+ve* and *-ve* index values are allowed) e.g.

```
squares.insert(-1, 81)
```

Negative indexing will insert it at the end of list. Positioning value to say where to insert the data in the list.

- Note: these *mutator* methods change the list but return the value `'None'`, also both of these could be achieved by assigning to *slices*

List method examples

- The `reverse()` method reverses the position of all elements in a list, e.g.

```
squares.reverse()
```

- The `remove()` method allows us to remove the first occurance of a specific element from a list, e.g.

```
squares.remove(100)
```

- The `clear()` method removes all elements from a list, e.g.

```
squares.clear()
```

- Remember: *mutators* change the list content, but return `'None'`, e.g.

```
squares = squares.reverse()  # squares would become 'None'
```

Do not do this

List method examples

Doesn't change anything just interacts with it.

- The `index()`, `count()`, and `copy()` methods are *accessors* rather than *mutators*, hence they DO NOT change the list but DO return a value
- The `index()` method returns the *index* of a specific element, e.g.

  would find something in a list

  ```
  squares.find(81)
  ```

- The `count()` method returns the count of how many times a specific element appears in the list, e.g.

  ```
  squares.count(36)
  ```

  Says how much of something there is
  ↳ Example, how long string is OR how many items in a list.

- The `copy()` returns a *shallow* copy of the list, e.g.

  ```
  squares_list = squares.copy()
  ```

Removing using del

Delete

→ Deletes. – Keyword in Python language

- The `del` statement can remove an element from a list using an index position, it is similar to `pop()` but does not return a value, e.g.

```
del a[0]        # remove the first element of the list
```

- The `del` statement can also remove a *slice*, e.g

```
del a[0:10]     # remove the first 9 elements

del a[:]        # remove all elements
```

- Finally, `del` can actually delete an entire variable, once a variable is deleted it cannot be accessed (unless recreated), e.g.

```
del a           # the variable 'a' no longer exists
```

List Comprehensions

Concise

- Up until now we have constructed our lists in a very “manual” way
- It is often the case however that list contents are calculated, or derived from other *iterable* values
- As an example, the *squares* list could be generated using code such as this -

```
squares = []
for x in range(10):
    squares.append(x*x)
```

This multiplies X by itself creating a squares list.

- A **list comprehension** allows us to write similar list initialisation code, but in a more concise manner

List Comprehensions

- A list comprehension uses square brackets (like normal lists) but includes an *expression* followed by `for` and optional `if` statements
- The list contents are created by evaluating the expression for each value iterated over within one or more for loops, e.g.

```
squares = [x * x for x in range(10)]
```

Instead of putting values in the [you put an expression.

- Within the above example `x * x` is the expression that is evaluated, for each value in range 0 to 9, it results in exactly the same list as the previous slide's code, i.e.

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List Comprehensions

A way of programmatically building up the contents of a list.

- Additional `if` statements can be used to restrict inclusion based upon certain conditions. e.g. to generate a list of squares from even numbers only

```
even_squares = [x * x for x in range(10) if (x % 2) == 0]
```

Requires multiple conditions to be true for it to execute.

This statement will omit any odd value from the list.

- Would result the list: `[0, 4, 16, 36, 64]`
- If multiple `for` statements are present, then these act like a nested loop, e.g.

```
times = [x * y for x in range(1,4) for y in range(1,4)]
```

- Is equivalent to the following regular code

```
times = []
for x in range(1,4):
    for y in range(1,4):
        times.append(x * y)
```

Introduction to Tuples

- A **Tuple** is a *built-in* **sequence** type, and is similar to a *List* in many ways
- Like a list, a Tuple consists of a number of ordered values

Used differently from a list because data of different types are in them.

- They have a slightly different syntax to lists, and are usually contained within *parentheses* (not always necessary but often wise!), e.g.

```
info = ("John", 85, 1.24)
```

→ uses ROUND brackets.

once created it cannot be changed - unlike a list.

- Tuples are often used in different situations to lists, and are **immutable**
- Tuples typically contain different types of values (in contrast to a list)
- The elements of a tuple are usually accessed by **unpacking** or **indexing**

Creating Tuples

- A tuple with no elements can be created using parentheses, e.g.

```
empty = ()
```

- A tuple containing a single element is created using a trailing comma, e.g.

```
singleton = 4,                  # parentheses are optional
```

- A multi-element tuple is created as a comma separated list, e.g.

```
info = "Eric", 82, 1.22         # parentheses are optional
```

- A tuple can be created from another *iterable* value, using the `tuple()` *constructor*, e.g.

```
copy = tuple(info)
```

Brackets are optinal in Tuples

Accessing Tuple Elements

- Creating a Tuple with multiple elements is known as **Tuple Packing**, e.g.

```
student = ("Terry, G.", 3.43, "Comp.Science")
```

- To access the elements we use **Sequence Unpacking**, e.g.

```
name, gpa, course = student
```

↳ selecting elements from a Tuple
Extracting values out of tuple

- This is often called *multiple assignment*, and is commonly seen in Python, but it is actually just *packing* and *unpacking*, e.g.

unpacking - assigns this. → Tuple

```
x, y, z = 10, 20, 30   # right-hand side is a tuple
```

Short hand way of assigning multiple variables.

- The value on the right-hand side can actually be any *sequence* type, including *lists*, *strings* etc. but in all cases the number of variables on the left-hand side *MUST* match the sequence *length*

Accessing Tuple Elements

Similar to list and String.

- Since a Tuple is a type of sequence, elements can be accessed using **indexing** and **slicing**, e.g.

```
module = (1, "Programming", 12, "Comp.Science")

name = module[1]
course = module[3]
short_mod = module[0:2]
```

- Remember however, a Tuple is **immutable**, so the following is NOT allowed

```
module[1] = "Python Programming"
```

- Some *methods* which do not mutate the Tuple are available, such as `count()` and `index()`

Tuples as arguments

used to pass arguments to a function

- The lesson describing functions mentioned the concept of specifying a *variable length argument* list (by prefixing the parameter with a '*'), e.g.

```
def concat(*args, sep="/"):
    return sep.join(args)
```

- Variable argument lists, are actually represented using tuples, e.g. to call the above we could do something like -

```
concat("join", "these", "together", sep="." )
```

- In this case we are passing a Tuple in the form -

```
("join", "these", "together")
```

Tuples as arguments

When calling functions you can pass tuples in to be unpacked.

- It is also possible to *unpack* a sequence prior to a function being called
- This is useful if values are within a sequence, but a function requires separate positional arguments e.g. given the following tuple -

```
module = (2, "Databases", 12, "Comp.Science")
```

- We could unpack this and call a function with separate arguments -

```
output_module(*module)
```

- The '*' prefix causes the tuple to be unpacked prior to the function call, and actually results in the call being made as follows -

```
output_module(2, "Databases", 12, "Comp.Science")
```

Lists vs Tuples

- Lists and Tuples are very similar but also have some differences. They are both are an ordered *sequence* of zero or more values, but -
  - Lists are **mutable**, whereas Tuples are **immutable**
  - Lists typically contain the same type of values (they're *homogeneous*),
  - Tuples typically contain different type of values (they're *heterogeneous*)
  - List elements are usually accessed via *iteration* or *indexing*
  - Tuple elements are usually accessed via *unpacking* or *indexing*

- **A general rule:** Tuples are typically used instead of a List when a small number of *differently typed* items need grouping as an *immutable* value

Summary

- **Lists** and **Tuples** are useful compound types
- Both support a number of **methods** that can be called
- **List comprehensions** allow sequences to be constructed programmatically
- Tuples are typically used to store small sequences of **heterogeneous** values
- **Tuple packing** and **Sequence unpacking** are used to create and access contained values
- Tuples are used to support **variable length argument** lists within functions. Sequences can be unpacked using '*****' during to function calls
- Lists are **mutable**, whereas Tuples are **immutable**