# Introduction to Programming

## Lab Worksheet

### Week 6

Prior to attempting this lab tutorial ensure you have read the related lecture notes and/or viewed the lecture videos on MyBeckett. Once you have completed this lab you can attempt the associated exercises.

You can download this file in Word format if you want to make notes in it.

You can complete this work using the Python interpreter in interactive mode. This could be inside an IDE, or just a command prompt.

**Topics covered:**

* Using List methods
* List Comprehensions
* Introduction to Tuples
* Packing and Unpacking

For more information about the module delivery, assessment and feedback please refer to the module within the MyBeckett portal.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## ©2021 Mark Dixon Modified 2021 Tony Jenkins

## **Using List Methods**

The List data-type provides a lot of very powerful functionality. A previous lesson introduced the basic ideas and syntax behind the List compound type. As a quick reminder:

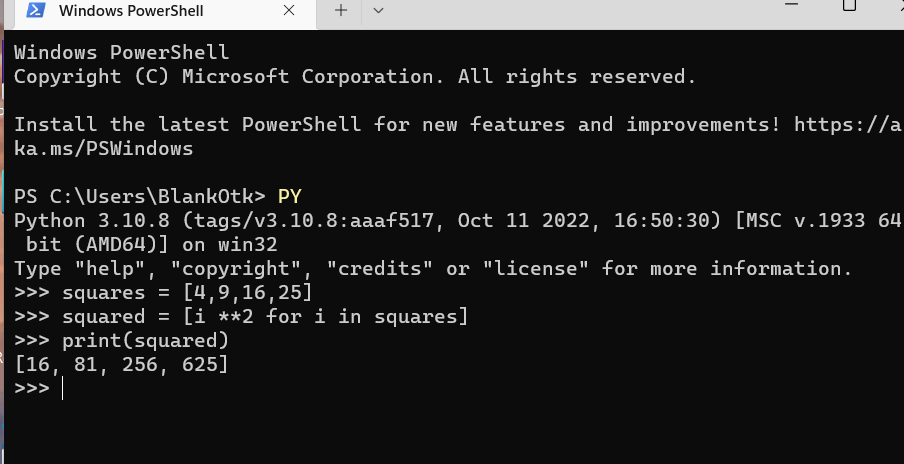
* A list is a *sequence* type that supports *indexing*, *slicing* and *concatenation.*
* A list is *iterable* and can be used in for…in type control statements.
* A list is *mutable*, so once created its contents can be changed.

The List type is built directly into the Python language hence there is a specific syntax associated with the creation and manipulation of lists. e.g. to create a list, square brackets can be used:

squares = [4, 9, 16, 25]

**TASK**: Write a for..in loop that iterates over all the elements of the squares list and prints the square root of each to the screen. *Hint*: you may want to import a function from the math module to help achieve this.

OUTPUT:



**Introducing Methods**

The term *method* has been occasionally mentioned during these lessons. A *method* can be thought of as being very similar to a *function* i.e. it is some predefined code that can be called while passing zero or more argument values. A method however is always associated with a specific data-type, and when called is prefixed with a value (often a variable) that gives the method *context*. What this means is that when the method executes it performs its operation using information associated with the value used during the call.

Since a method is associated with a data-type, different data-types provide different methods, which implies that methods available on one data-type are not necessarily available on other data-types.

The List data-type has a number of methods that allow access and manipulation of the list. One example is the append() method we have previously seen, e.g.

squares.append(36)

Notice how the method call is prefixed with the variable name, followed by a period (.).

The fact that the variable squares data-type is a *List* makes this method call valid, since the method append() is defined for any value based on the *List* data-type.

**TASK**: Write some code that uses the append() method to add the next three square values (49, 64, 81) to the end of the squares list.

There are many other methods defined for the List type. Some of these methods *mutate* (change) the list contents, others just *access* the contents. Many of these methods provide operations that can also be achieved using *slicing*.

The methods that ***mutate*** a list are -

extend() - appends multiple values, by providing an *iterable* argument value

insert() - inserts a single value at a specific index position

remove() - removes the first occurance of a specific element

pop() - removes and returns the last element added to the list

clear() - removes all elements from the list

sort() - sorts the contents of the list

reverse() - reverses the position of all elements in the list

The methods that only ***access*** the list are -

index() - returns the index of a specific element

count() - returns the count of how many times a specific element appears

copy() - returns a shallow copy of the list

The built-in help system can provide information about methods which are available on a specific data-type. For example, a list of the methods available on the List type can be viewed by inputting the following command. You can ignore the methods that start with an underscore for now.

OUTPUT:

Text

Description automatically generated

help(list)

**The extend() method**

The extend() method mutates (changes) a list by adding multiple values at once. It is similar to the append() method, but takes a sequence as a parameter, like so:

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

If the same parameter had been passed to the append() methods, then the list itself would have been appended as a value, instead of each individual value being extracted and appended individually.

Note: the above command could also be achieved by assigning to a slice as follows (but the code is probably a little more cumbersome and opaque).

squares[len(squares):] = [49, 64, 100]

**TASK**: Write some code that uses the extend() method to add the next three square values, starting at 121 (11 x 11), to the end of the squares list.

OUTPUT:

Text

Description automatically generated

The built-in help system can also provide information about methods, simply by prefixing the method name with the type name. So to get help about the extend method you can type -

help(list.extend)

If you are ever unsure of what parameters need to be passed to a method, just use the help() command to find out.

**The insert() method**

The insert() method mutates a list by inserting a value at a specific position within the list. The position is specified as a zero-based *index* value. As with other indexing, it is possible to use a negative index value to identify an insertion position relative to the end of the list.

The insert() method takes two arguments, the first is the index position, the second is the value to be inserted.

**TASK**: Write some code that uses the insert() method to insert the value 2, to the very beginning of the squares list.

OUTPUT:

Text

Description automatically generated

Note: As with many of these methods slicing could have been used as an alternative. So to insert a single value the equivalent would be as below. As before, the code using the method is probably a little clearer.

squares[index:index] = [value]

**The remove() method**

The remove() method mutates a list by removing a specified value from the list. The value to be removed is provided as the single parameter, if the value is not present within the list then an error is reported. If the same value appears more than once within the list, then only the first occurence found is removed.

**TASK**: Write some code that uses the remove() method to remove the value 49 from the squares list. Print the list afterwards to ensure the value has indeed been removed.

OUTPUT:

Text

Description automatically generateds

**TASK**: Write some code that uses the remove() method to remove the value 3 from the squares list. Notice how an error is generated since the given value was not present.

OUTPUT:

**Text

Description automatically generated**

**TASK**: Create a simple list that contains the values [1, 2, 3, 1, 2] and then use the remove () method to remove the value 2. Which value is removed?

OUTPUT:

Text

Description automatically generated

**The pop() method**

The pop() method mutates a list by removing and returning the last (right-most) value from the list. The term pop is well known in computer science and it refers to removing the last element that has been added to a *stack* type structure.

**TASK**: Write some code that uses the pop() method to remove and display the last value of the squares list. Print the list afterwards to ensure the value displayed has been removed.

OUTPUT:

Text

Description automatically generated

The pop() method can optionally be provided with an *index* parameter. If this is present then the value is removed from the specified position within the list, rather than the end. Typing the following command will describe how this parameter is optional.

help(list.pop)

As with most index values, a negative value can be used to make the position relative to the end of the list. Also if the index is out of range (so larger than the length of the list), an error is reported.

**TASK**: Write some code that uses the pop() method to remove and display the first value of the squares list. Print the list afterwards to ensure the value has been removed.

Note: It might seem as if remove and pop are doing the same thing (especially when pop is used with an index value), but there is a difference. Review the descriptions above if you cannot see what that is!

OUTPUT:

Text

Description automatically generated

**The clear() method**

The clear() method mutates a list by removing all elements. After the method call the list still exists, but it is empty.

>>> some\_list = [1,2,3,4]

>>> some\_list.clear()

>>> print(some\_list)

[]

clear() is a very simple method and equivalent to the following *slicing* assignment -

some\_list[:] = []

Again, using the method is probably better practice as the resulting code is easier to understand.

**The sort() method**

The sort() method mutates a list by changing the order of the elements. This is probably the most complex of all the List type methods to use (in some cases). It is also probably the most powerful.

If called with no parameters this sorts the list into ascending order by comparing the value of each element.

>>> some\_list = [8, 2, 4, 99, 1]

>>> some\_list.sort()

>>> print(some\_list)

[1, 2, 4, 8, 99]

This list was sorted numerically, since it contained integer type values. If it had contained string type values then it would have been sorted alphabetically.

The sort method can become more complex to use because it can optionally take *keyword-arguments*. One of these is a reverse boolean value indicating whether the sort should be done in reverse:

>>> some\_list = [8, 2, 4, 99, 1]

>>> some\_list.sort(reverse=True)

>>> print(some\_list)

[99, 8, 4, 2, 1]

The other keyword-argument is a key which specifies a function that can be used to customize the applied sort order. This argument is typically provided as a *lambda expression*, which was discussed in a previous lesson.

The provided function can return an alternative value that is to be used within the sort algorithm. This sounds rather complex, but looking at an example should help simplify things a little.

Imagine you had a list of people’s names that needed sorting. This could be achieved very easily using the following code -

>>> names = [ "Mark", "Alicia", "Ben" ]

>>> names.sort()

>>> print(names)

['Alicia', 'Ben', 'Mark']

In this case the names have been sorted alphabetically. However, what if you wanted to sort the list based on the length of each name? This sounds difficult, but it can be done fairly easily by providing a l*ambda expression* as the key argument and using the len function to return the lengths of strings.

>>> names.sort(key=lambda n : len(n))

>>> print(names)

['Ben', 'Mark', 'Alicia']

This works since the *lambda expression* returns the length of each name, using a call to the len(n) built-in function. The sort() method calls the given function for each name in the list, then uses the returned value as the sorting criteria rather than the name itself.

**TASK**: Write some code that uses the sort() method with no arguments, to alphabetically sort the exact list of names shown below. Display the list after the sort has been called.

names = [ "Eric", "anna", "Sophie", "sam" ]

Once you have completed the previous task you should have noticed that the order is not what you may have expected. The result probably looked like this -

['Eric', 'Sophie', 'anna', 'sam']

This is because lower-case letters appear later than all upper-case letters when a default sort is applied.

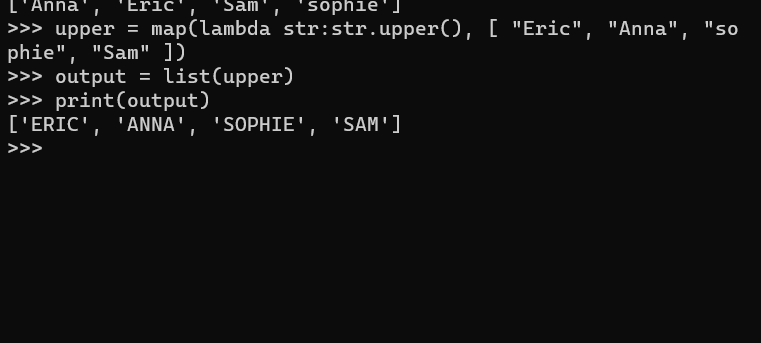
OUTPUT:

Text

Description automatically generated

**TASK**: Improve your previous solution so that the list is sorted correctly, ignoring the case used to write the names. To achieve this you will have to include a key argument in the form of a *lambda expression* that returns each string as uppercase letters only. Hint: you can use the str.upper() method to convert a name to uppercase letters.

OUTPUT:



There is also a built-in *function* called sorted() that can also be used to sort lists. Unlike the sort() *method* however this produces a new List in memory, rather than sorting the current list *in-place*. Therefore the sorted() function does not mutate the list itself, but produces a new one altogether. This is useful if the original list order is NOT to be changed, but a sorted copy is needed.

>>> some\_list = [8, 2, 4, 99, 1]

>>> sorted\_list = sorted(some\_list)

>>> print(some\_list)

[8, 2, 4, 99, 1] # order of original list is unchanged

However, in the case where the original list can be changed it is much more efficient to sort a list using the sort() method rather than the sorted() function. e.g. the following lines of code both do exactly the same thing, but the first version is more efficient since it is sorting the original list *in-place*, rather than creating a brand new list.

some\_list.sort() # sort in-place

some\_list = sorted(some\_list) # sort using function, then assign

The sorted() function can take the same *keyword arguments* as the sort() method, so is just as powerful. It can also be applied to any *iterable* type. which means it can sort more than just lists.

**The reverse() method**

The reverse() method mutates a list by reversing the position of each element.This is done *in-place* so it is fairly efficient, i.e. it does not need to create a new temporary variable or list.

**TASK**: Write some code that uses the reverse() method to reverse the values of the squares list. Print the list afterwards to ensure the values have been reversed.

OUTPUT:

Text

Description automatically generated

Note: A reverse operation could be done using *slicing* as shown below. However, once again it is probably more intuitive to call the reverse() method instead.

squares[:] = squares[::-1]

**The index(), count(), and copy(), methods**

The index(), count(), and copy() methods are *accessors* rather than *mutators*, hence they do not change the list but always return a value.

The index() method returns the index of a specific element. If the element is not present then an error is reported.

>>> colours = ["red", "green", "yellow", "blue", "red"]

>>> print(colours.index("yellow"))

2

An optional *start* and *end* parameter can be specified to limit the range of the list to be searched. These act as index values between which the search is to be performed (from *start* to *end*-1), e.g.

>>> colours = ["red", "green", "yellow", "blue", "red"]

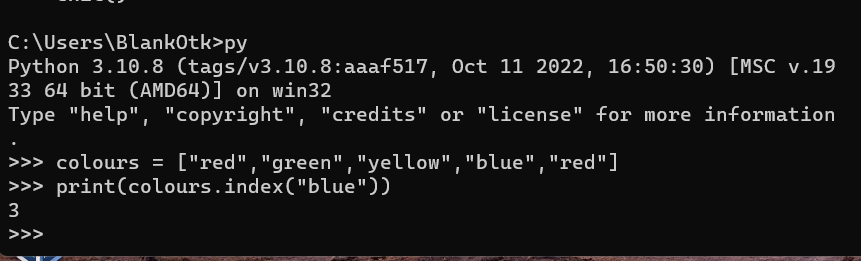
>>> print(colours.index("red",3))

4

This example finds the second occurrence of "red" (at index 4) since it starts searching from position 3 rather than the beginning of the list.

**TASK**: Write some code that finds the index of the colour blue.

OUTPUT:



The count() method returns the count of how many times a specific element appears in the list:

>>> print(colours.count("red"))

2

>>> print(colours.count("black"))

0

Finally, the copy() method returns a *shallow* copy of the list, e.g.

>>> new\_colours = colours.copy()

>>> print(new\_colours)

['red', 'green', 'yellow', 'blue', 'red']

People new to programming often make the mistake of trying to take a copy of a list by using an assignment, like this:

>>> new\_colours = colours

>>> print(new\_colours)

['red', 'green', 'yellow', 'blue', 'red']

Although this looks to have worked, it has not actually created a copy of the list; it has merely created a second variable that refers to the *same* list. Hence, if the original list is changed then the new\_colours list will also be changed, since there is only really one underlying list. This is shown here:

>>> new\_colours = colours

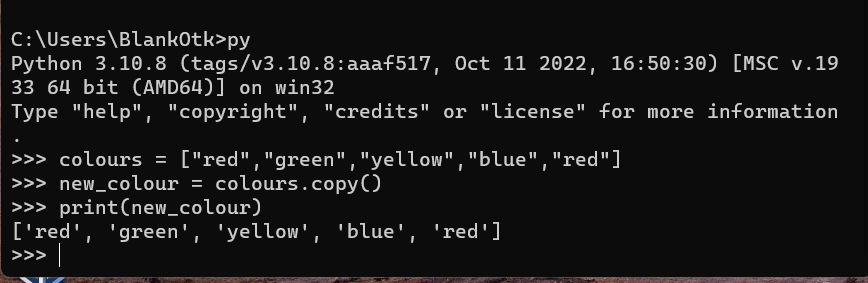
>>> colours.append("black")

>>> print(new\_colours)

['red', 'green', 'yellow', 'blue', 'red', 'black']

**TASK**: Write some code that makes a copy of the colours using the copy() method. Then make some changes to the original list. Print the contents of the copied list to ensure these changes have not affected the copy.

OUTPUT:



**The del Statement**

The del statement is not specific to lists, but a general statement available within the Python language. It can be used to delete values from lists using both *indexing* and *slicing*. It can also be used to delete entire variables. Examples of its use are:

>>> del colours[0] # remove first colour

>>> del colours[-1:] # remove last colour

>>> del colours # remove the colour variable

The del statement is much more destructive than the mutator methods. Once a variable is deleted it cannot be accessed (unless recreated).

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## **List Comprehensions**

Lists can be created in multiple ways. Up until now most lists have been created by explicitly providing values within the code. In many cases however lists are constructed programmatically. They are generated by the program itself.

For example, it would be very easy to generate a squares list using a for...in loop along with the append() method. The code might be:

squares = []

for x in range(10):

squares.append(x\*x)

An alternative to writing such explicit code to populate a list also exists, this is known as a *list comprehension*. They achieve the same outcome but in a more concise manner. Using them is very *Pythonic*.

*List comprehensions* are *expressions* that appear instead of the values within a list initialisation statement. The above example could be rewritten using a list comprehension as follows:

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

The expression (x \* x) is evaluated for each iteration of the subsequent for...in loop, then the result is placed within the list. The result is neater and probably easier to understand.

**TASK**: Write some code that uses a list *comprehension* to create a list called cubes that contains the cubed values (x \* x \* x) of the numbers from 2 to 20 inclusive.

OUTPUT:

Text

Description automatically generated

It is also possible to further restrict inclusion of values by using an additional if statement to the end of the for...in loop. Only when the if statement returns True does the value get evaluated and added to the list. So to generate a list containing all the even numbers between 100 and 200 the following list comprehension could be used:

even\_nums = [num for num in range(100,201) if num % 2 == 0]

The if statement is particularly useful when creating lists which are subsets of other lists. For example, if a list existed called all\_users then the following code could be used to create a subset of that list.

some\_users = [u for u in all\_users if len(u) < 8]

**TASK**: Examine the above code and work out which user names will be placed in the some\_users list. What is the condition that has to be met for inclusion.

It is even possible to have multiple for...in loops and multiple if conditions within the same list comprehension, although this can get complicated to understand. If more than one for...in loop exists then this acts like a *nested loop* where the inner loop is executed for every iteration of the outer loop.

>>> triples = [n for n in range(1,6) for count in range(3)]

>>> print(triples)

[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5]

The above example adds the numbers 1,2,3,4 and 5 to the list three times each.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## 

## **Introduction to Tuples**

A *Tuple* is the name given to a built-in data-type that is similar to a List. It is a compound type that contains a *sequence* of ordered values.

However, Tuples are often used in different situations to lists, and are *immutable*. They also typically contain different types of values, in contrast to a list that typically contains the same type of values. Since a Tuple is a sequence, like a *List* and a *String*, it supports *indexing*, *slicing*, *concatenation* and is *iterable* (it can be used within for...in loops).

Tuples are initialised using a slightly different syntax to lists, with the initial values being contained within parentheses (round brackets):

student = ("Griffin, P.", 2, 38.2)

Creating tuples in this way is often known as *Tuple packing*.

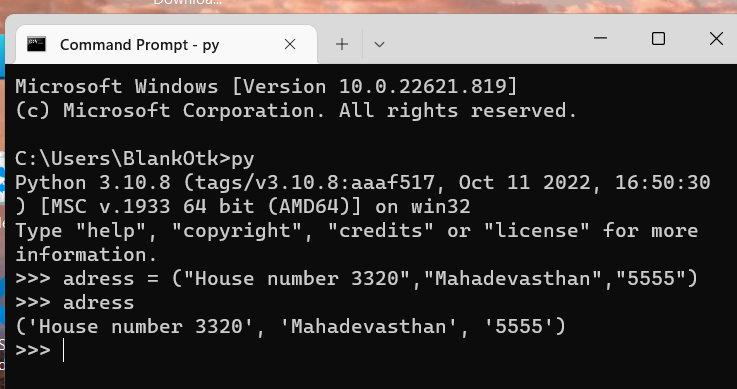
Tuples are often used to represent information or “*records*”. What the values in a tuple actually represent is upto the programmer and depends on their use within the program. In the above example, the values may well represent “name”, “year of study”, and “average grade”.

Tuples can actually be created without the use of parentheses, as below, but it is more usual to include the brackets.

student = "Griffin, P.", 2, 38.2

**TASK:** Create a tuple called address that includes your own “house number”, “street” and, “postcode” as three different values.

OUTPUT:



**Empty and Single element Tuples**

An empty tuple can be created just using parentheses -

empty = ()

To create a tuple with only one element, a trailing comma must be used, e.g.

the\_one = "Neo",

This is a slightly odd syntax, but a common error is to try to create a one element tuple as follows:

the\_one = ("Neo")

This actually just creates a *string* type variable, not a tuple.

**TASK:** Try entering the above examples to create single element tuples. Then use the type() function to examine the data-type of the created variables.

OUTPUT:

Text

Description automatically generated

Hint: For this reason it is common to include a trailing comma even when one is not needed. For example:

student = ("Griffin, P.", 2, 38.2,)

**Sequence Unpacking**

As already stated the elements within a tuple can be accessed in the same way as a list, by using *indexing*, *slicing*, or *iteration*. For example:

>>> print(student[0])

'Griffin, P.'

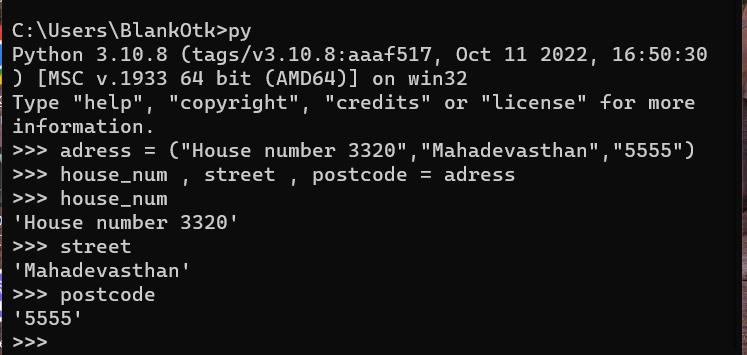
However, it is also common to access the elements using a technique called *sequence unpacking*. This works like this:

name, year, average\_grade = student

The values within the tuple on the right-hand side are assigned (in order) to the variables present on the left-hand side. Hence the number of values on the left-hand side MUST match the number of values within the given tuple.

**TASK:** Use *sequence unpacking* to extract the values you stored within the address tuple earlier. Unpack the tuple into variables named house\_num, street and postcode.

OUTPUT:



*Sequence unpacking* is also sometimes called *multiple assignment* and is commonly used to quickly assign multiple values to multiple variables using a single statement:

x, y, z = 10, 20, 30

Notice how the number of values on the right-side matches the number of variables on the left-side.

This is actually just using *tuple packing*, followed by *sequence unpacking*. As the name suggests *sequence unpacking* can be applied to any type of *sequence*. Hence the right-hand side can also be either a *list* or a *string*, although this is less commonly seen.

Using *tuple packing* and *unpacking* is also a convenient way of returning more than one value from a function. When used well this is an especially useful feature of Python, and is not something that is available in many programming languages.

def calc\_squares(x, y):

return ( x \* x, y \* y )

a,b = calc\_squares(42, 92)

A tuple can also be unpacked when calling a function or method that takes a variable number of arguments. A ‘\*’ prefix causes a tuple to be unpacked prior to the function call:

my\_range = (10, 30)

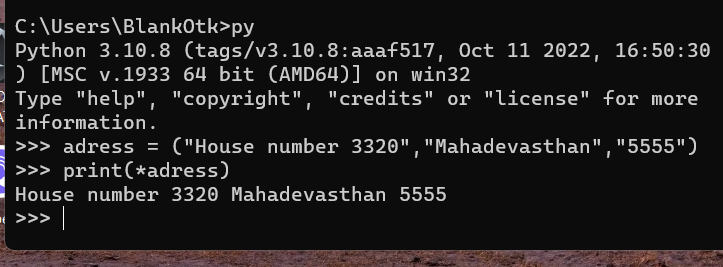
for i in range(\*my\_range):

print(i)

This unpacks the my\_range tuple during the call, resulting in the equivalent to range(10, 30), since the ‘\*’ causes each tuple element to be passed as a separate argument.

**TASK:** Write some code that calls the print() function to output the contents of the address tuple you created earlier. Ensure you use the ‘\*’ prefix so that the elements are extracted before being passed to the function. Compare this with a version of the same code that calls the print() function without using the ‘\*’ prefix,

OUTPUT:



**Tuple Methods**

Since tuples are *immutable* they have fewer *methods* than the List type. The reason is simply that methods that change the content are not applicable, so methods such as append(), extend() and insert() do not exist since the tuple content cannot be changed after creation.

In fact the only methods that are available are count() and index(), which are accessors. These work in exactly the same way as they do with the list type.

Also, since tuples are *immutable* it is not possible to assign to *indexes* or *slices*. e.g. the following code is not possible on tuples -

student[0] = "Griffin, Brian"

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## 

## 

## **Key Terminology**

Understanding key terminology is important not only when programming but for understanding computer science in general. Certain terms, technologies and phrases will often re-occur when reading or learning about many computer science topics. Having an awareness of these is important, as it makes learning about the various subjects and communication with others in the field easier.

**TASK**: Look at each of the phrases below and ensure you understand what each of these means. For any that you do not understand, do a little research to find a definition of each term. This research may involve looking back over these notes, or the associated lecture notes. It may also involve searching for these terms on the Internet.

* Method
* List comprehension
* Tuple
* Tuple Packing
* Sequence Unpacking

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## **Practical Exercises**

You have now completed this tutorial. Now attempt to complete the associated exercises.