3. Then `cd` followed by your username to enter your user folder.

   We are now in: `C:\Users\<username>`

4. Then `cd Desktop` to enter your Desktop folder.

   We are now in: `C:\Users\<username>\Desktop`

5. Then finally `cd` followed by your project folder name to enter the project folder you created on your desktop.

   We are now in: `C:\Users\<username>\Desktop\<project-folder-name>`

Navigating step by step this way is just an alternative way of navigating like you would in a File Explorer. In the File Explorer, If we started in the C drive we could double click the Users folder to move there, then your username folder, then your Desktop folder, and so on.

▼ `cd` on MacOS or Linux

For example, if your project folder is on your Desktop, you could enter: `cd /home/<username>/Desktop/<project-folder-name>`.

Alternatively, you could take it step-by-step and enter:

1. `cd /` to enter your root folder

   We are now in: `/`

2. Then `cd home` to enter the home folder within your root folder

   We are now in: `/home`

3. Then `cd` followed by your username to enter your user folder

   We are now in: `/home/<username>`

4. Then `cd Desktop` to enter your Desktop folder

   We are now in: `/home/<username>/Desktop`

5. Then finally `cd` followed by your project folder name to enter the project folder you created on your Desktop

   We are now in: `/home/<username>/Desktop/<project-folder-name>`

If you are getting errors, make sure to check for typos. The directory you enter is case sensitive so make sure words folders with capitals like Users and Desktop are capitalised.

Once you think you have reached your folder containing your Python script, entering `dir` (Window) or `ls` (MacOS and Linux) will **show you the files inside that folder**. If you see the name of your Python script (helloworld.py), you are in the correct folder.

▼ Example on Windows

I have created a folder called "python-for-beginners" on my desktop that holds a file called helloworld.py

When I open the terminal, I can navigate to this folder using the command `cd C:\Users\TomDraper\Desktop\python-for-beginners`

To create a **Boolean** (true or false) variable, you can either give a variable a value of `True` or `False` (the first character must be capitalised):

```
x = True
```

```
x = False
```

Booleans are useful when representing whether the state of something is 'on' or 'off', such as whether you are currently logged in to an account, or whether your Bluetooth is enabled.

The name 'Boolean' is capitalised as it is named after the mathematician George Boole.

Comments

In the code snippets that follow, and throughout the rest of this course, comments will be used to try to explain and clarify code. Comments are found in most programming languages and are plain English notes that the programmer can use to try and explain what their code is doing. They are completely ignored by Python and have no effect on what your program does. You could delete comments from your code, and it will work exactly the same.

Comments in Python are written as a **hashtag** followed by your comment. For example:

```
# This is a comment
```


You can write comments on a line of their own, or after a piece of code.

```
# This is a comment
x = 76 # This is another comment
```

It's important to note that everything from the hashtag character onwards until the end of a line is considered part of the comment. You are therefore unable to write a comment before a piece of code on the same line. The code below would not have any effect:

```
# This is a comment x = 76
```

Here, the entire line (including the `x = 76`) is considered a comment and the `x = 76` would **not** be executed by Python.

Multiline Comments

If you want to write a long comment that spans multiple lines, Python can do multi-line comments by containing it between a pair of **triple inverted commas** `'''`. All text between these triple inverted comments is considered a comment. Although, the hashtag version is often preferred, the multiline comment is typically reserved to describe the purpose of a large chunk of code.

```
w = not True
x = (2 == 10) and (5 <= 100)
y = not (x or w)
z = (15 > 50) or y
```

What would be assigned to each of the variables `w`, `x`, `y`, `z`? `True` or `False`?

▼ Answer

Variable `w` is `False`

Variable `x` is `False`

Variable `y` is `True`

Variable `z` is `True`

Variable `w`:

`not True` equals `False`

Variable `x`:

`(2 == 10) and (5 <= 100)` equals `False and True` which equals `False`

Variable `y`:

`not (x or w)` equals `not (False or False)` which equals `not False` which equals `True`

Variable `z`:

`(15 > 50) or y` equals `False or True` which equals `True`

Programming In Python

Control Structures

Programming languages use various 'control structures' to **control the flow of execution**. Typically, when code begins running, it begins executing lines of code one after another. Control structures allows the code to jump and divert to different areas of the code based on whether a given condition is satisfied.

The two main control structures within Python are called 'if statements' and 'loops'. These are common programming structures and found within most languages. **If statements** allow you to execute a particular chunk of code **if and only if a given condition is satisfied**. **Loops** allow you to **repeatedly execute the same chunk of code** until a condition becomes satisfied.

If Statements

If statements are a type of control structure. They can divert the flow of execution and stop Python from simply executing line-after-line sequentially. They offer a way to produce dynamic behaviour within your program.

If statements create a structure such that a given **block of code** is only executed **if** a given **condition** is evaluated to `True`.

If

An **if statement** in Python uses the `if` keyword, followed by the **condition** you want to check, followed by a **colon**. If this condition is found to be `True`, the block of code below condition will be executed. For Python to know that this code block is **within the if statement**, it must be **indented.** This means that each line of the code block should start with **four spaces,** but two spaces would also work. This choice of indentation size is completely up to the programmer, as long as you remain consistent with the size chosen.

```
if condition:
    # Execute this indented block of code
    # ...

# Continue with the rest of the program
# ...
```

Your text editor can often be configured to insert the correct indentation with a single click of the tab key.

▼ **Note:** The tab key

A tab character is actually its own unique character and not just made up of multiple space characters. To avoid having to repeatedly press the spacebar four times to create indentation, text editors for programming are often configured to **just produce four spaces** instead of a tab character **when the tab key is pressed**.

An often cause of frustration with Python is that it will produce an error when tab characters and space characters are mix-and-matched within indentation, even if they appear to be the same size. If you are getting indentation errors when your indentation appears to be consistent and correct, you may have pressed the tab character and actually inserted a **tab character**, so check the tab key configuration within your text editors to ensure it only produces spaces.

For example:

```
month = 'December'
day = 25

if (month == 'December') and (day == 25):
    print('Merry Christmas!')
    print('And a Happy New Year!')
```

The **if statement** in the code above checks whether `month` is equal to `'December'` AND whether `day` is equal to `25`. Python will check this condition and evaluate it to either `True` or `False`. It will print the Christmas greeting to the screen only if this condition is found to be `True`.

The program initially assigns the `month` and `day` variables with the value `'December'` and `25`. As a result, and when this program is run the condition will be evaluated to `True` and it will print:

> Merry Christmas!
> And a Happy New Year!

If the month were changed to `'February'`, the condition `(month == 'December') and (day == 25)` would be `False` and nothing would be printed to screen.

It only makes sense for the condition within an if statement to contain a variable, as this allows for dynamic behaviour because the evaluation of the condition can change if the value of the variable changes. If the if statement is run at multiple points in the program, sometimes it may execute the code block, and other times it may ignore it as the values of variables change throughout the life of the program.

get chance to review the condition `age > 18`, even though this is `True`. Therefore, the order of your `elif` conditions is important and can affect the result.

Exercise - Odd or Even

You are given a variable that can hold a number. Write an if statement that checks the value of the number to see whether it is odd or even.

If it is odd, your program should print 'odd'.
If it is even, your program should print 'even'.

Copy the code below containing the number variable to use.

```
number = 503

# Code to check whether number is odd or even
# If number is even print 'even', otherwise print 'odd'
# ...
```

▼ Hint 1

We can check whether a value is even using the modulus operator `%`.
Remember: `A % B` results in **the remainder once A has been cleanly divided into B.**

▼ Hint 2

If `number % 2` is equal to 0, it means `A` can be cleanly divided by 2, with **no remainder left over**. This means `number` must be **even**.

If `number % 2` is equal to 1, it means `A`, is divided by 2, it leaves a remainder of 1 left over. This means `number` must be **odd**.

▼ Answer

With the example number of 503, your program should print 'odd', but it should adapt when this number is changed.

Answer 1

We can use `number % 2` to check whether `number` is odd or even.

- If `number % 2` results in 0, `number` divides cleanly by 2, and therefore it is **even**.
- If `number % 2` results in 1, `number` **does not** divide cleanly by 2, leaving a remainder of 1, and therefore it is **odd**.

We can use an if statement to check whether number is **even** first, and if not, we can use an `elif` statement to check whether it is **odd**.

```
number = 503

if number % 2 == 0:
    print('even')
elif number % 2 == 1:
    print('odd')
```

```
year = 2016

# Code to check whether the year is a leap year
# ...
```

▼ Hint 1

We can check to see if the year is perfectly divisible using the modulus operator `%`. If the year modulus a value is equal to 0, the year is perfectly divisible by that value.

▼ Answer

There are many different solutions to this problem. All of them have slight advantages and disadvantages.

Answer 1

We could check the two conditions in turn, but this is a bit repetitive, and the computer may waste time evaluating `(year % 4) == 0` and `(year % 100) == 0` two separate times. Although the computer will process our program extremely fast and in practice we would never notice if our program was taking very slightly longer due to unnecessarily repeating computations. It's good practise to try and avoid redundant computations when you can.

We also have identical results for two of the if statements code blocks: `print(year, 'is a leap year')`. It's good practise to try and avoid repeating your code, as it can make your program more difficult to understand. It also makes your code easier to maintain as you remove the risk of altering a line of code but forgetting to alter the code in **all** places where that same line is used. There is almost always a way to eliminate duplicate lines and make your code more concise.


```
year = 2016

if (year % 4) == 0 and (year % 100) == 0 and (year % 400) == 0:
    # Perfectly divisible by 4, 100 and 400
    print(year, 'is a leap year')
elif (year % 4) == 0 and (not (year % 100) == 0):
    # Perfectly divisible by 4, but not 100
    print(year, 'is a leap year')
else:
    print(year, 'is not a leap year')
```

Answer 2

We could use an `OR` to join the two conditions that make a leap year. But the computer still has to unnecessarily evaluate `(year % 4) == 0` and `(year % 100) == 0` twice. The condition also becomes very long and difficult to read.

```
year = 2016

if ((year % 4) == 0 and (year % 100) == 0 and (year % 400) == 0) or ((year % 4) == 0 and (not (year % 100) == 0)):
    # Perfectly divisible by 4, 100 and 400
    print(year, 'is a leap year')
else:
    print(year, 'is not a leap year')
```

Within the loop we may want to know the number of times we have looped so far. Is this the first time we are executing this code? Or the tenth? This is where the 'for loop' **variable** comes in. The 'for loop' automatically creates a new variable for us to use within this loop. With each loop, this variable will be **assigned the current loop number**. In programming languages, whenever we are in a situation that involves counting, it always **begins from zero**. This can be confusing at first and will take a while to get used to. The 'for loop' variable follows this rule. During the first loop it will hold the value 0, during the second it will hold 1, and so on. Because we begin from 0, during the last loop of the code block it will hold the value `N-1`. We can use this variable to produce different behaviour within the code block depending on the number of times we have looped. Just like any variable, we can give it any name we like, although quite often this variable is given the name `i`.

If we wanted to create a 'for loop' that loops 5 times, we would write:

```
for i in range(5):
    print(i)
```

This will print:

> 0
> 1
> 2
> 3
> 4

The 'for loop' variable `i` takes the values from 0 to 4, but this is still 5 different values, so the loop has looped **5 times**.

On the first loop, `i` first takes the value 0. It then prints this value and returns to the top of the 'for loop' to get incremented by 1. Then it executes the same code block again, but this time `i` holds a value of 1. This continues until `i` is **no longer less than 5** and it exits the 'for loop' and continues with the rest of the program.

If we wanted to, we could correct the fact that the loop counter begins at zero, and instead print the numbers 1 to 5 by adding 1 to `i` each time we print.

```
for i in range(5):
    print(i+1)
```

This will print:

> 1
> 2
> 3
> 4
> 5

Starting Value

We can also specify an optional starting value.

- Tuples
- Sets

and each of them have different properties and uses.

Previously our variables would hold a **value** that had a **data type**, such as holding the value 4, which is an integer. We have now seen that our variables can hold **a value that is a data structure** (which holds other values). These Python data structures (list, dictionaries, tuples, and sets) are in fact data types themselves, just like integer, float, string or Boolean. They just happen to be data types that have the ability to **hold other data types**.

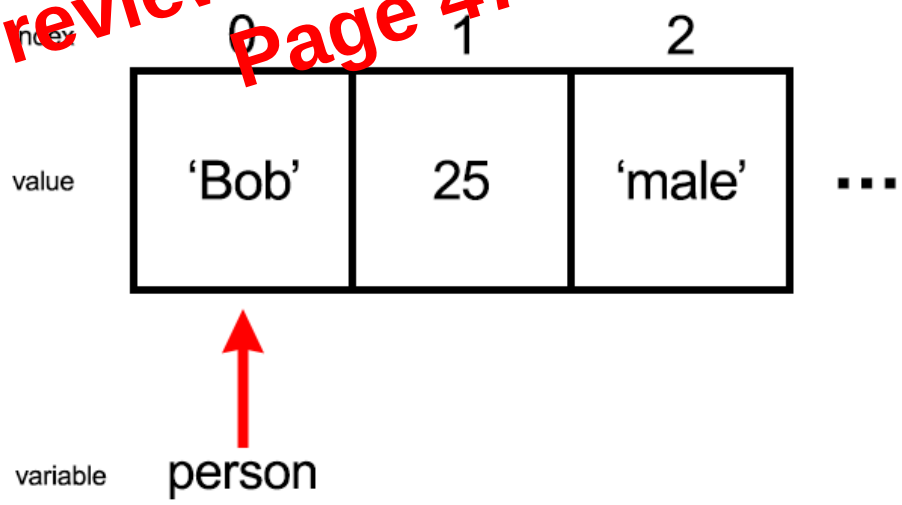
Lists

Lists (sometimes referred to as **arrays**) are the first data structure in Python that we will cover. Just like a shopping list, a Python list just holds a series of values.

A list can:

- hold any number of values
- add or remove items at any time
- hold values of any data type

When you create a Python list, you create a **sequence of storage spaces** in memory called **elements**. Each of these elements can hold a **value of any data type**. To be able to access a particular space, each of these spaces are labelled by an numerical **index** value. The first space has an index of 0, the second space has an index of 1, and so on.



In the example above, we represent a person as a list of their values. Although the `person` variable points to the first element of the list (`'Bob'`), it can access any value by using its index. At index 0 of the list is the person's name, at index 1 is their age and their gender is at index 2. If we wanted to, we could extend the list and create a space at index 3 to hold another value.

Creating a List

Python uses the square brackets `[]` to create a list of items.

```
person = ['Bob', 25, 'male']
```

Alternatively, we can create an **empty list** and add values later. We can create an empty list using an empty set of square brackets `[]`. We can add items to the end of our list using `person.append()` with the value to append within the parenthesis.

```
person = []  # Create an empty list and assign it to the person variable
print(person)
person.append('Bob')
print(person)
person.append(25)
print(person)
person.append('male')
print(person)
```

Once run, this code will print:

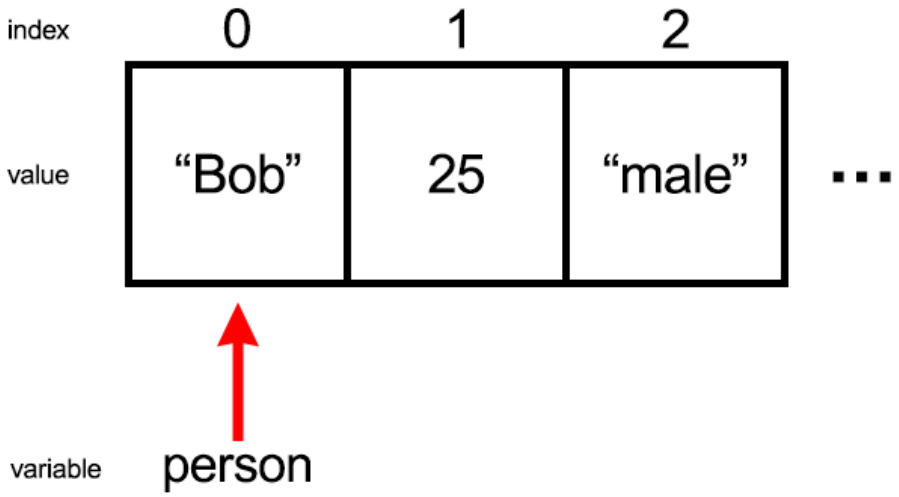
> []
> ['Bob']
> ['Bob', 25]
> ['Bob', 25, 'male']


```
person = ['Bob', 25, 'male']
print(type(person))
```

If we print out the type of `person`, it will display `<class 'list'>`. This confirms that a 'list' is in fact a Python data type.

Accessing Elements

Once we have a list containing values, we can access these values by using the index of a specific value.




```
person = ['Bob', 25, 'male']

print(person[0])
print(person[2])
```

Once run, this code would print:

> 'Bob'
> 'male'

List Length

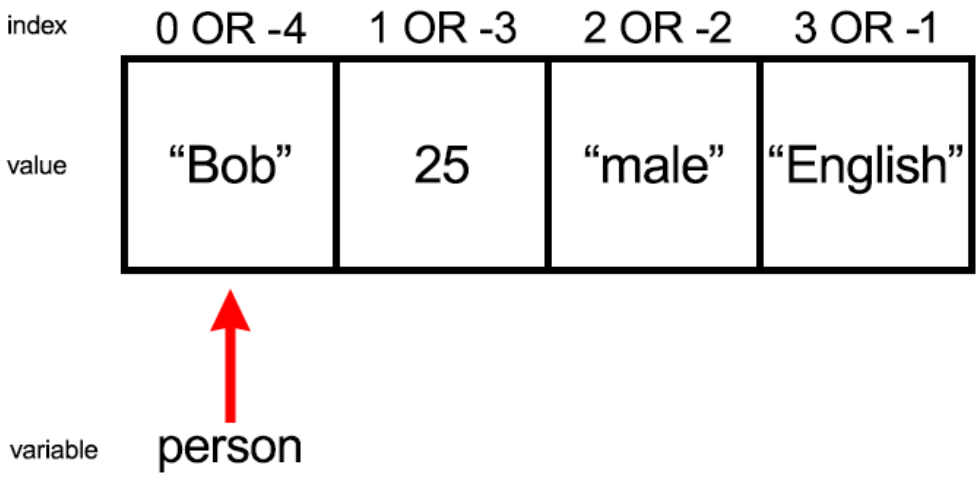
Python can calculate the length of a list using `len()`, short for 'length'.

```
colours = ['red', 'blue', 'green']

list_length = len(colours)  # Assign length value to list_length
print(list_length)

# Add two new items to the colours list
colours.append('orange')
colours.append('purple')

list_length = len(colours)  # Assign the new length value to list_length
print(list_length)
```

Once run, the code above will print:

> 3
> 5

To access the final element in a list, we would need to know the index of that value. If we do not know its index, we could calculate it using the length of the list.



This is convenient way of accessing the last few elements without having to first calculate the length of the list using `len()`.

```
person = ['Bob', 25, 'male']

print(person)
print(person[2])
print(person[-1])

person.append('English')

print(person)
print(person[2])
print(person[-1])
```


One run the code above will print:

> ['Bob', 25, 'male']
> 'male'
> 'male'
> ['Bob', 25, 'male', 'English']
> 'male'
> 'English'

Initially, the value `'male'` can be accessed by **both index 2 and index -1**. Once another value is appended to the end of the list, index 2 still accesses `'male'`, but **index -1 accesses the new value added to the end of the list**.

Index Out of Bounds

When using an index, you need to make sure that the index you use is **valid**.

With list of **3 elements**, only indexes between **0 to 2, or -3 to -1** are valid, any higher or lower and you would **run off the end of the list** and you would get an **error**.

```
person[0] = 'Wendy'
print(person)

# Access the last element and change it to be 'female'
person[-1] = 'female'
print(person)
```

Once run, the code above will print:

> ['Bob', 25, 'male']
> ['Bob', 34, 'male']
> ['Wendy', 34, 'male']
> ['Wendy', 34, 'female']

Iterating Over Values

We can now combine loops with lists to iterate over values within a list.

With a list containing **5 elements**, we can use a '**for loop**', with `range(5)`, to loop **5 times**. The loop will run each time, and the variable `i` will be assigned the 5 values, from **0 to 4**. This happens to perfectly match the possible indices of the list and we can use variable `i` to access the value at each index from 0 to 4 and immediately print it out.

```
my_list = [10, 20, 30, 40, 50]

for i in range(5):
    print(my_list[i])  # Print the value at index i
```


Once run, this code would print:

> 10
> 20
> 30
> 40
> 50

Although often, we may not immediately know the length of a list, or the length of the list may change, so it is better to replace 5 with the length of the list, using `len(my_list)`.

```
my_list = [10, 20, 30, 40, 50]

for i in range(len(my_list)):
    print(my_list[i])  # Print the value at index i
```

This would produce the same output.

For Each Loops

'For loops' offer an alternative way to loop over values of a list. Rather than using `range(N)` to give our counting variable `i` the value of numbers up to (not including) N, we can replace it with the **actual list variable** (e.g.,

```
list_sum = 0
for i in range(20):
    # Access list element at index i
    # Increase list_sum by the value at this index
    list_sum += my_list[i]

print(list_sum)
```

Answer 2

Alternatively, we could use a 'for each loop' to loop through each **value within the list,** without needing to access each value using its index. With each loop, the `value` variable will take the **next value within the list**. We can then add this value to the `list_sum` variable.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

list_sum = 0
# Iterate over each value in the list
# Each time store it in the variable value
for value in my_list:
    # Increase list_sum by this value
    list_sum += value

print(list_sum)
```


Exercise - Find Value

You're given an extensive list of numbers. Write a program to check whether a certain value is within this list. Print 'Yes' if so, otherwise print 'No'.

Copy and complete the code below containing the list, and the value to check for.

```
numbers = [871, 143, 297, 849, 128, 574, 755, 133, 331, 448, 147, 168, 996,
           459, 722, 481, 239, 674, 101, 256, 477, 633, 901, 898, 208, 649, 435, 596,
           831, 230, 893, 408, 969, 180, 141, 422, 417, 511, 623, 899, 267, 871, 639,
           126, 262, 804, 269, 526, 401, 689, 590, 645, 785, 949, 638, 954, 746, 445,
           877, 839, 878, 946, 740, 443, 737, 334, 559, 573, 790, 794, 146, 639, 114,
           132, 730, 757, 954, 791, 257, 292, 809, 215, 931, 895, 682, 435, 739, 684,
           716, 415, 880, 144, 638, 959, 206, 967, 269, 245, 368, 305]

value_to_check_for = 730

# Check whether 730 is within the numbers list
# ...
# Print 'Yes' or 'No'
```

▼ Hint 1

We could loop over each value within the list and check each value to see if it matches.

▼ Hint 2

Once the loop has finished, we will need to have remembered whether we found the value we were searching for. We could create use a separate Boolean variable that is initially set to `False` before the loop begins, and set to `True` within the loop when we find the value we are searching for.

▼ Answer

Your program should print `'Yes'`. There are many viable solutions to this problem, but the method below is the most straightforward.

We use a Boolean variable `value_to_check_for` to be the indicator as to whether we have seen the value we are searching for. We scan through the list, checking whether each number is equal to the value to check for. If a number is found, we set the `value_to_check_for` to be `True`, otherwise `value_to_check_for` will never change from being `False`. Once we have finished looping through the list, we can simply check whether `value_to_check_for` is `True` or `False` using an if statement to give our 'Yes' or 'No' answer.

```
numbers = [871, 143, 297, 849, 128, 574, 755, 133, 331, 448, 147, 168, 996,
           459, 722, 481, 239, 674, 101, 256, 477, 633, 901, 898, 208, 649, 435, 596,
           831, 230, 893, 408, 969, 180, 141, 422, 417, 511, 623, 899, 267, 871, 639,
           126, 262, 804, 269, 526, 401, 689, 590, 645, 785, 949, 638, 954, 746, 445,
           877, 839, 878, 946, 740, 443, 737, 334, 559, 573, 790, 794, 146, 639, 114,
           132, 730, 757, 954, 791, 257, 292, 809, 215, 931, 895, 682, 435, 739, 684,
           716, 415, 880, 144, 638, 959, 206, 967, 269, 245, 368, 305]

value_to_check_for = 730

# Initially set found_number to False
# If we find the number, we can change this to True
found_number = False

# Loop through each number in the numbers list
for n in numbers:
    # Check if the number 'n' is equal to 730
    if n == value_to_check_for:
        # If so, remember that we've found the number!
        found_number = True

# Print the result
if found_number:
    print('Yes')
else:
    print('No')
```


List Filter

There may be a situation where there is a certain type of item within a list that you no longer need and want to remove. We could build a program that checks each item within a list to see whether it meets a certain condition and deletes the item from the list if it does. This would essentially 'filter out' any items that do not meet the condition.

Filtering by Data Type

The items within a list can be a mixture of different data types. We could build a program that loops over each item in a list, checks whether each item is a certain data type (e.g., **string** by checking whether `type(item) == str`), and remove it if so.

The program below filters out any strings that are contained within the list.

```
my_list = ['red', 5.2, 10, 'orange', True, 'green', None, 14, 1.0, 'blue']

length_of_list = len(my_list)

# Loop through each index value of my_list
```

```
for i in range(length_of_list):
    item = my_list[i]  # Get the item in the list at this index

    if type(item) == str:  # Check if the item is a string
        del my_list[i]  # Remove the item from the list
```

However, we run into two problems when removing items this way:

1. When we use `del my_list[i]` we remove the item from the list, creating a gap in the list. To close this gap, Python automatically moves all other following items one space to the left. For example, if we find a string at index 2 and remove this item, the items at indices **3, 4, 5, 6...** must shift across one space to become **2, 3, 4, 5...**. This means the next item we need to check has been **moved from index 3 to index 2**. This means that during our next loop, our loop counting variable `i` will be **incremented to become 3**, and we would have **skipped the next value which is now at index 2**. We needed to have remained at the **same index**, because that's where the next item will be after removing an item.
2. With a `for` loop, the proposed values that `i` will take during the loop are calculated a single time, **at the point that the for loop is first entered**. If we have a **list of 5 items**, and create a loop to check each index from **0 to 4**, if at some point we remove an item, the items to the right will be shifted across one space to fill the gap created, and the **length of the list will decrease by one**. This means that the **final index of the list changes from 4 to 3, and 4 is no longer a valid index**. However, we have already told the loop we are looping until `i` has taken the value of 4, and during the final loop, **index 4 of the list will be checked** (which no longer exists), which will cause an **error**.

```
person = ('Bob', 25, 'male')
print(type(person))
```

If we print out the type of `person`, it will display `<class 'tuple'>`.

Accessing Values

We can access tuple values in exactly the same way as a list.

```
person = ('Bob', 25, 'male')

print('The name is:', person[0])
print('The age is:', person[1])
print('The gender is:', person[2])
```

Once run, the code above will print:

> The name is: Bob
> The age is: 25
> The gender is: male

Iterating Over Values

We can iterate over the values within a tuple in exactly the same way as we can with a list.

```
person = ('Bob', 25, 'male')

for value in person:
    print(value)
```

Once run, the code above would print:

> Bob
> 25
> male

Slices

Slices of tuples are also the same as lists, except a slice of a tuple results in a new tuple, rather than a list.

```
python_letters = ('p', 'y', 't', 'h', 'o', 'n')

print(python_letters[2:])  # From index 2 onwards
print(python_letters[1:3])  # From index 1 to index 2
```

Once run, the code above would print:

Once run, the code above would print:

> {'blue', 'purple', 'orange', 'green', 'red'}
> {'purple', 'orange', 'green', 'red'}

.discard()

We can discard a named value from the set using `.discard()`, with the value within the parentheses. This works exactly like `.remove()`, however **if the value entered is not within the set, this will NOT cause an error.**

Like with lists, Python can check for the presence of a given item is in a set using the `in` keyword. The code `value in my_set` will return `True` if `value` is in the set named `my_set`, and `False` otherwise.

```
colours = {'red', 'blue', 'green', 'purple', 'orange'}

print(colours)
print('blue' in colours)  # Check whether 'blue' is in colours - True or False

colours.discard('blue')

# 'blue' is no longer in the colours set
print(colours)
print('blue' in colours)
```

Once run, the code above would print:

> {'blue', 'purple', 'orange', 'green', 'red'}
> True
> {'purple', 'orange', 'green', 'red'}
> False

.pop()

If we want to **retrieve and remove** a random item from the list, we can use `.pop()` with no value within the parentheses. `.pop()` will remove a value from the set and return it to you for you to assign to a variable or print out.

```
colours = {'red', 'blue', 'green', 'purple', 'orange'}

print(colours)

# Retrieve a value from the set and assign it to the colour variable
colour = colours.pop()

print('The colour we have popped is:', colour)
print(colours)
```

Once run, the code above would print:

> {'blue', 'purple', 'orange', 'red', 'green'}
> The colour we have popped is: blue
> {'purple', 'orange', 'red', 'green'}

Iterating Over Values

Just like with lists or tuples, we can loop over the values within a set.

```
colours = {'red', 'blue', 'green', 'purple', 'orange'}

for colour in colours:
    print(colour)
```

Once run, this code would print:

> red
> blue
> green
> purple
> orange

Set Operations

Sets have their own operators that can take two sets and perform an action to create a new set.

Intersection `&`

The intersection operation will take two sets, A and B, and return **a set containing only the values that appear in both A and B**.

```
set_a = {2, 4, 6, 8, 10}
set_b = {1, 2, 3, 4, 5}

print(set_a & set_b)
```


Once run, the code above would print `{2, 4}`.

Union `|`

The union operation will take two sets, A and B, and return **a set containing the values that appear in either A or B or both**.

```
set_a = {2, 4, 6, 8, 10}
set_b = {1, 2, 3, 4, 5}

print(set_a | set_b)
```

Once run, the code above would print `{1, 2, 3, 4, 5, 6, 8, 10}`.

Difference `-`

The different operation will take two sets, A and B, and return **a set containing the values within set A, with all the values that are also in set B removed**.

expert has already produced a great solution that you could use. You can simply download a solution and import the file into your code file using the `import` keyword, without having to manually copy and paste the parts that you need.

A collection of related code files that can be imported and used within a program is called a 'library'. There are Python libraries for every possible type of application. Python has libraries for everything from data visualisation and graphing to image manipulation to libraries that can make web page requests.

The Python Standard Library

Python has a collection of official inbuilt libraries, called the **Python Standard Library**, which come preinstalled when you install Python. These code files will live somewhere within the directory that Python is installed and can be easily imported into any program you write.

These libraries include:

- `random` - a library that generates random numbers
- `math` - a library that contains lots of mathematical functions
- `datetime` - a library for creating and dealing with dates and times
- `csv` - a library for reading and writing .csv files

The full list and information about each library can be found here:

> The Python Standard Library - Python 3.9.5 documentation
>
> While The Python Language Reference describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.
>
> https://docs.python.org/3/library/index.html

Any library from The Python Standard Library can be easily imported by simply entering `import` followed by the library name at the top of your code file.

For example:

```
# Import all of the functions from within the math.py library
from math import *

# Import all of the functions from within the random.py library
from random import *

# Code using the math and random libraries
# ...
```

Installing a Package with Pip

Python libraries made by members of the community can be installed using a tool called 'pip'.

'pip' is a **package manager** for Python, and it can be used to fetch and download a specific library onto your machine for you to use in your code.

This course will **not** require any libraries that must be installed on to your device using pip. However, you're highly likely to require pip if you continue to use Python beyond a beginner level. Some basic instructions to install and use pip can be found below.

- After the second use of `sort_pass(numbers)`, the **two largest numbers** are in their final sorted positions.
- After the third use of `sort_pass(numbers)`, the **three largest numbers** are in their final sorted positions, **which means the final fourth number must also be in its final position.**

If we have a list of N numbers, we only have to use `sort_pass(numbers)` **N-1 times** until we have a completely sorted list. On the N-1th time, placing the next largest value into it's correct place will also force the smallest value into it's correct position on the far left.

▼ Answer

If we say the numbers list has N numbers, to sort the list we would have to run the `sort_pass` function N-1 times.

- On the first use, the largest number will be moved into place
- On the second use, the second largest number will be moved into place
- On the third use, the third largest number will be moved into place
- On the (N-1)th use, the (N-1)th largest number will be moved into place

All numbers except one will have been moved into their final sorted position, which means the last number would also have to be in its sorted position!

If we apply the `sort_pass` function any more than N-1 times, the list would still be fully sorted. The `sort_pass` function would simply check each number, see they are all already in the correct order and not make any switches.

We can find the length of the list using `len(numbers)`.

```
def sort_pass(numbers):
    for i in range(len(numbers) - 1):
        # If two numbers next to each other are the wrong way around
        if numbers[i] > numbers[i+1]:
            # Swap the two numbers at index i and i+1
            temp = numbers[i+1]
            numbers[i+1] = numbers[i]
            numbers[i] = temp
    return numbers

# Define a function to sort a list of numbers
def sort(numbers):
    # Use sort_pass (len(numbers) - 1) times
    for i in range(len(numbers) - 1):
        numbers = sort_pass(numbers)  # Sort the next biggest number
    return numbers


numbers = [5, 1, 9, 8, 3, 2, 6]
sorted_numbers = sort(numbers)
print(sorted_numbers)
```


Before You Go

Book Recommendations

Automate the Boring Stuff with Python - Al Swelgart