Hyperiondev

TASK

Data Types and Conditional Statements

Visit our website

Introduction

WELCOME TO THE DATA TYPES AND CONDITIONAL STATEMENTS TASK!

In this task, you'll start to learn about data types, conditional statements, and logical operators.

The two most fundamental data types are strings and numeric data types. Strings are some of the most important and useful data types in programming. Why? Let's think about it like this: When you were born, your parents did not immediately teach you to do mathematical sums, not 1 + 1 or standard deviation. The first thing they taught you to do was to speak, to say words, to construct full sentences, to say "Mom" or "Dad". Well, this is why we are going to 'teach' the computer to first be able to communicate with the user – and the only way to do this is to have a good grasp of strings.

Next, we consider numbers. We encounter them daily and, in most cases, multiple times a day. It is likely that, through studying mathematics at school, you were introduced to different types of numbers such as integers and decimal numbers, as well as the operations we can perform on them, such as addition, subtraction, multiplication, and division. Computer programming languages like Python provide support for storing and manipulating many different types of numbers. You will learn how numbers are categorised based on their nature, as well as how to perform arithmetic operations in Python.

With conditional statements, we gain the ability to guide our code's decisions, adjusting its behaviour according to specific conditions. This dynamic attribute elevates our programs, making them more responsive and adaptable. Finally, we'll give some thought to operators – symbols that tell the compiler or interpreter to perform specific behaviours (whether it be mathematical, relational, or logical) and produce a final result.

WHAT ARE STRINGS?

A *string* is a list of letters, numerals, symbols, and special characters that are put together. The values that we can store in a string are vast. An example of what strings can store is the surname, name, address, etc. of a person.

In Python, strings must be written **within** quotation marks (**" "**) for the computer to be able to read them.

The smallest possible string contains zero characters and is called an *empty string* (i.e. `string = ""`).

Examples of strings:

```
name = "Linda"

song = "The Bird Song",

licence_plate = "CTA 456 GP"
```

Strings are probably the most important data type in programming. They are used as a medium of communication between the computer and the user. The user can enter information as a string, and the program can use the data to perform calculations and finally display the calculated answer to the user.

```
name = "John"

joke = "Knock, knock, Who's there?"
```

You can use any name for your variable, but the actual string you are assigning to the variable must be within these **""** (quotation marks).

Defining multi-line strings

Sometimes, it's useful to have long strings that can go over one line. We use triple single quotes (**''' '''**) to define a multi-line string. Defining a multi-line string preserves the formatting of the string.

For example:

```
long_string = ''' This is a long string

 using triple quotes preserves everything inside it as a string

 even on different lines and with different \n spacing. '''
```

STRING FORMATTING

Strings can be added to one another. In the past, we used to use a method called **concatenation**, which looked like this:

```
name = "Peter"

surname = "Parker"

full_name = name + surname
```

`full_name` will now store the value “PeterParker”.

The `+` symbol simply **joins** the strings. If you wanted to make your code more presentable, you could put spaces between the words.

```
full_name = name + " " + surname
```

We now added a blank space in between the two strings, so `full_name` will now store the value “Peter Parker”. **Note: you cannot concatenate a string and a non-string**. You need to **cast** the non-string to a string if you want to concatenate it with another string value. If you try to run code that adds a string and a non-string, you will get an error. For instance, if we wanted to add an age of 32, we would have to cast it as a string to print it.

```
print(full_name + str(32))
```

But this is a clunky way of formatting strings. This way of putting together a string is still used in older languages, such as Java, and does have its place, but it’s much better practice to use the `format()` method.

```
name = "Peter Parker"

age = 32

sentence = "My name is {} and I'm {} years old.".format(name, age)

print(sentence)
```

In the example above, a set of opening and closing curly braces (`{}`) serve as a placeholder for variables. The variables that will be put into those placeholders are listed in the brackets after the keyword `format`. The variables will fill in the

placeholders in the order in which they are listed. Therefore, the code above will result in the following output: `My name is Peter Parker and I'm 32 years old.`

Notice that you don't have to cast a variable that contains a number (`age`) to a string when you use the `format()` method.

F-strings

The shorthand for the format function is **f-strings**. Take a look at the example below:

```
name = "Peter Parker"

age = 32

sentence = f"My name is {name} and I'm {age} years old."

print(sentence)
```

In f-strings, instead of writing `.format()` with the variables at the end, we write an `f` before the string and put the variable names within the curly brackets. This is a neat and concise way of formatting strings. When formatting strings, f-strings are often preferred for their readability and simplicity.

NUMBERS AS STRINGS

We can even store numbers as strings. When we are storing a number (e.g. 9, 10, 231) as a string, we are essentially storing it as a word. The number will lose all its number-defining characteristics. So the number will not be able to be used in any calculations. All you can do with it is read or display it.

In real life, sometimes we don't need numbers to do calculations, we just need them for information purposes. For instance, the house number you live in (let's say: 45 2nd Street UK 2093) won't be used for performing any calculations. What benefit would there be for us to find out what the sum is of all the house numbers in an area? The only thing we need is to know what number the house is when visiting or delivering a package. The number just needs to be visible.

This is the same concept when storing a number as a string; all we want is to be able to take in a value and display it to the user.

For example:

```
telephone_num = "041 123 1234"
```

ACTIONS THAT CAN BE PERFORMED ON STRINGS

Working with strings in Python is both powerful and intuitive. Strings offer a variety of operations that can be performed on them, enhancing their functionality and usability. Here are some of the key actions you can perform on strings:

- **Indexing**: Access individual characters in a string.
- **Slicing**: Extract a substring from a string.
- **Extended slicing**: Extract a substring with a specific step.
- **Using string methods**: Utilise built-in methods to manipulate and analyse strings.

You can use `len()` to get the number of characters in a string or the actual length of a string. The `len()` function is a general function that works with various data types, including strings, to return the number of items in an object. The print statement below prints out 12, because "Hello world!" is 12-characters long, including punctuation and spaces.

```
print(len("Hello World!"))
```

Output:

```
12
```

You can also **slice** a string. Slicing in Python extracts characters from a string, based on a starting index and ending index. It enables you to extract more than one character or "chunk" of characters from a string. The print statement below will print out a piece of the string. It will start at position/index **1** and end at position/index **4** (which is not included).

```
greeting = "Hello"

print(greeting[1:4])
```

Output:

```
ell
```

You can even use negative numbers inside the brackets. The characters are indexed from right to left using negative numbers, where **-1** is the rightmost index and so on. Using negative indices is an easy way of starting at the end of the string instead of the beginning. This way, **-3** means "third character from the end".

Look at the example below. The string is printed from the third-to-last character, "l", all the way to the end of the string. Notice that you don't need to specify the end of the index range:

```
greeting = "Hello"

print(greeting[-3:])
```

Output:

```
llo
```

In the example below, the slice begins from position `0` and goes up to, but not including, position `1`. This is because if the starting position is left blank, it will by default start at `0`.

```
greeting = "Hello"

print(greeting[:1])
```

Output:

```
H
```

In the next example, the slice begins from position `1`, includes position `1`, then continues to the end of the string and skips/steps over every other position. This is known as an extended slice. The syntax for an extended slice is `[begin : end : step]`. If the end is left out, the slice continues to the end of the string.

```
greeting = "Hello"

print(greeting[1::2])
```

Output:

```
el
```

In this final example, you can think of the '`-1`' as a reverse order argument. The slice begins from position `4`, continues to position `1` (not included), and skips/steps backwards one position at a time:

```
greeting = "Hello"

print(greeting[4:1:-1])
```

Output:

```
oll
```

You can print a string in reverse by using `[::-1]`. Remember that the syntax for an extended slice is `[begin : end : step]`. By not including a beginning and end, and specifying a step of `-1`, the slice will cover the entire string, backwards, so the string will be reversed. You can find out more about extended slices **here**.

Note: slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable.

```
new_string = "Hello world!"

fizz = new_string[0:5]

print(fizz)

print(new_string)
```

Output:

```
Hello
Hello world!
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

METHODS TO MANIPULATE STRINGS

Most programming languages provide **built-in** functions to manipulate strings, i.e., you can concatenate strings, search within strings, extract substrings, and perform various other operations. There is more to be learned about this topic, but for now we will stop here. If you're very keen to look ahead, you can find a number of resources online to **learn about string methods**. **Hint**: this resource will come in handy for your first auto-graded task later on.

Now, let's move on to learning about numeric data types.

NUMBERS IN PYTHON

There are three distinct numeric data types in Python: integers, floating-point numbers, and complex numbers.

- **Integers**: these are synonymous with whole numbers. Numbers which are stored as this type do not contain a fractional part or decimal. Integers can either be positive or negative, and are normally used for counting or simple calculations. For instance, you can store the number of items you wish to purchase from a store as an integer, e.g. `num = int("-12")`.
- **Floats**: decimal numbers or numbers which contain a fractional component are stored as floats. They are useful when you need more precision, such as when storing measurements for a building or amounts of money. Floats may also be in scientific notation, with `E` or `e` indicating the power of 10, e.g. `x = float("15.20")`, `y = float("-32.54e100")`.
- **Complex**: complex numbers have a real and imaginary part, which are each a floating-point number, e.g. `c = complex("45.j")`.

DECLARING NUMBERS IN PYTHON

When you declare a variable, Python will already know if it is a **float** or an **integer** based on its characteristics. If you use decimals, it will automatically be a float and if there are no decimals, then it will be an integer.

```
class_list = 25    # integer

interest_rate = 12.23    # float
```

CASTING BETWEEN NUMERIC DATA TYPES

To cast between numbers, make use of the `int()` or `float()` functions, depending on which is needed.

```
num1 = 12

num2 = 99.99

print(float(num1))

# Converting floats to ints, as below, causes data loss. int() removes values

# after the decimal point, returning only a whole number.

print(int(num2))
```

ARITHMETIC OPERATIONS

Doing calculations with numbers in Python works similarly to the way they would in regular maths.

```
total = 2 + 4

print(total)  # Prints out 6
```


```
total = 2 + 4

print(total)  # Prints out 6
```


The only difference between calculations in real mathematics and programming is the symbols you use:

Arithmetic operations	**Symbol used in Python**
Addition: adds values on either side of the operator.	**+**
Subtraction: subtracts the value on the right of the operator from the value on the left.	**–**
Multiplication: multiplies values on either side of the operator.	*****
Division: divides the value on the left of the operator by the value on the right.	**/**
Modulus: divides the value on the left of the operator by the value on the right and returns the remainder.	**%**
Exponent: performs an exponential calculation, i.e. calculates the answer of the value on the left to the power of the value on the right.	******

MATHEMATICAL FUNCTIONS

In Python, numerical data types play a crucial role in performing mathematical operations and calculations. To work with numerical data effectively, Python offers a variety of built-in functions (pre-written code) and libraries that can be utilised. Built-in functions are readily available within Python and provide fundamental mathematical operations, while the `math` module (a file containing Python code) needs to be imported (explicitly loaded into your program) and offers an extensive collection of specialised mathematical functions.

Some of the most commonly used functions are listed in the tables below.

Built-in mathematical functions:

Function	Description	Example

`round()`	Rounds a floating-point number to the nearest whole number, or decimal places as specified by the second argument.	`number = 66.6564544`
		`print(round(number,2))`
		# Will output `66.66`

`min()`	Returns the smallest value from an iterable, such as a list or tuple.	`numbers_list = [6,4,66,35,1]`
		`print(min(numbers_list))`
		# Will output `1`

`max()`	Returns the largest value from an iterable, such as a list or tuple.	`numbers_list = [6,4,66,35,1]`
		`print(max(numbers_list))`
		# Will output `66`

`sum()`	Calculates the total sum of all elements in iterable, such as a list or tuple.	`numbers_list = [6,4,66,35,1]`
		`print(sum(numbers_list))`
		# Will output `112`

Mathematical functions available through the `math` module:

To use the functions in the `math` module, add this line of code to the **top** of your program:

```
import math
```

Function	Description	Example
`math.floor()`	Rounds a number down.	`print(math.floor(30.3333))` # Will output `30`
`math.ceil()`	Rounds a number up.	`print(math.ceil(30.3333))` # Will output `31.0`
`math.trunc()`	Cuts off the decimal part of the float.	`print(math.trunc(30.33333))` # Will output `30`
`math.sqrt()`	Finds the square root of a number.	`print(math.sqrt(4))` # Will output `2.0`
`math.pi()`	Returns the value for pi where pi is the number used to calculate the area of a circle.	`print(math.pi)` # Will output `3.141592653589793`

Feel free to explore some more functions in the `math` module **here**.

IF STATEMENTS MAKE DECISIONS WITH CODE

Let's consider how we make decisions in real life. When you are faced with a problem, you have to see what the problem entails. Once you figure out the crux of the problem, you then follow through with a way to solve this problem. Teaching a computer how to solve problems works in a similar way. We tell the computer to look out for a certain problem and how to solve the problem when faced with it.

We are now going to learn about a vital concept when it comes to programming: We will be teaching the computer how to make decisions for itself using something called an *if statement*. As the name suggests, this is essentially a question. *If statements* can compare two or more variables or scenarios, and perform a certain action based on the outcome of that comparison.

If statements contain a condition, as you can see in the example below. Conditions are statements that can only evaluate to *True* or *False*. If the condition is *True*, then the indented statements are executed. If the condition is *False*, then the indented statements are skipped.

In Python, *if statements* have the following general syntax:

```
if condition:

    indented statements
```

Here's a code example of a Python *if statement*:

```
num = 10

if num < 12:  # Don't forget the colon!

    print("the variable num is lower than 12")
```

This *if statement* uses the `<` (less than) operator to check if the value of the variable `num` is less than 12. If it is, then the program will print the sentence letting us know. If `num` was greater than 12, then it would not print out that sentence. This is a slightly contrived example for the sake of simplicity, because we already know `num` is less than 12, having assigned it the value 10.

However, imagine that the value of `num` was input at run-time by a user; in that case, we could not know in advance what they would enter, and the condition the *if statement* evaluates would offer more value. Note that as all user input is assigned the data type *string*, we need to cast the input to an `int` if we want it to perform operations based on its value. See the example below:

```
num = int(input("Please input a number between 1 and 100"))

if num < 12:  # Don't forget the colon!

    print("The value you entered is lower than 12")
```

Notice the following important syntax rules for an *if statement*:

- In Python, the colon is followed by code that can only run if the statement's condition is *True*.
- The indentation (where each line of code is situated in relation to the surrounding code) is there to demonstrate that the program's logic will follow the path it indicates. Every indented line will run if the program's *if statement*'s condition is *True*.
- In Python, when writing a conditional statement such as the *if statement* above, you have the option of putting the condition in brackets if you'd prefer to (i.e. `if (num < 12):`). While your code will still run without the brackets, using brackets can help with readability when your code gets more complicated, which is a very important consideration in coding.

COMPARISON OPERATORS

You may have also noticed the less than (`<`) symbol in the previous code examples (`if num < 12:`). As a programmer, it's important to remember the basic logical operators. We use comparison operators to compare values or variables in programming. These operators work well with *if statements, if-else statements*, and *loops* to control what goes on in our programs.

The four basic comparative operators are:

- greater than `>`
- less than `<`
- equal to `==`
- not `!`

Take note that the symbol we use for **equal to** is '`==`', and not '`=`'. This is because `==` literally means 'equals' (e.g., `i == 4` means `i` is equal to `4`). We use `==` in conditional statements. On the other hand, `=` is used to assign a value to a variable (e.g. `i = "blue"` means I assign the string `blue` to `i`). It is a subtle but important difference.

As you can see, the *if statement* is pretty limited as is. You can only really make decisions that result in one of two possible outcomes. What happens if we have more options? What if one decision will have further ramifications and we will need to make more decisions to fully solve the problem at hand?

Control structures are not limited to *if statements*. You will soon learn how to expand on an *if statement* by adding an *else statement* or an *elif statement* (else if). First, however, let's find out about the boolean data type.

WHAT ARE BOOLEANS?

Booleans were developed by an English mathematician and computer pioneer named George Boole (1815–1864). A boolean data type can only store one of two values, namely *True* or *False*. One byte is reserved for storing this data type.

We use booleans when checking if one of two outcomes is *True*. For instance: is the car insured? Is the password correct? Is the person lying? Do you love me?

Once the information is stored in a variable, it's easy to use *loops* and *if statements* to check an extensive sample of items and base your calculations on the result of a boolean value.

ASSIGNING BOOLEAN VARIABLES

Assigning a boolean to a variable is very simple. You declare the variable name and then choose its starting value. This value can then be changed as the program runs.

For example:

```
pass_word = False

pass_word = True
```

BOOLEANS IN CONTROL STATEMENTS

Control statements allow you to use booleans to their full potential. How can you use a boolean value once you have declared it? This is where the *if statement* comes into play. Let's look at a simple decision we might make in our everyday lives.

If it is cold outside, you would likely wear a jacket. However, if it is not cold, you might find a jacket unnecessary. This is a type of branching: if one condition is true, do one thing, and if the condition is false, do something else. This type of branching decision-making can be implemented in Python programming using *if–else* and *if-elif-else* statements.

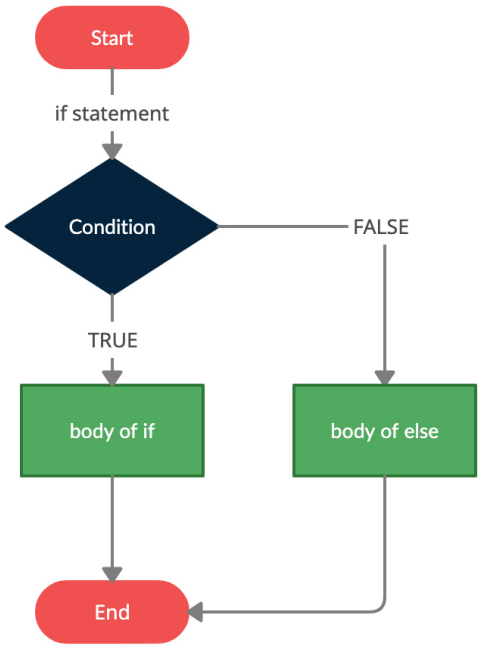
Let's try another example. When you are about to leave your house, do you always take an umbrella? No, you only take an umbrella if it is raining outside. Although this is another very rudimentary example of decision-making where there are only two outcomes, keep in mind that we can apply these basic principles to create more complex programs. Here's the scenario represented in code:

```
umbrella = "Leave me at home"

rain = False

if rain:

   umbrella = "Bring me with"
```

Take a look at line three in the example above. That is shorthand for **`if rain == True:`**, but because the default of a boolean is *True*, to write all of that is redundant, so we only use **`==`** with boolean conditionals if a condition specifically needs to be *False*.

THE STRUCTURE OF IF-ELSE STATEMENTS

In addition to the basic *if statement* functionality, it is possible to check the condition supplied to the *if statement* and not only do something if it evaluates to *True*, but do something else if it is *False*, as we discussed when talking about branching decisions. This is called an *if-else statement*. The basic structure of an *if-else statement* can be represented by this diagram:



Flowchart (Toppr, 2021)
https://www.toppr.com/guides/python-guide/tutorials/python-flow-control/if-elif-else/python-if-if-else-if-elif-else-and-nested-if-statement/

The example below extends the earlier *if statement* example to include *else*:

```
num = int(input("Please input a number between 1 and 100"))

if num < 12:  # Don't forget the colon!

    print("The value you entered is lower than 12")

else:

    print("The value you entered is higher than 12")
```

ELIF STATEMENTS

The last piece of the puzzle when it comes to *if statements* is called an *elif statement*. Elif stands for "**else if**". With this statement, you can add more conditions to the *if statement*, and in doing so, you can test multiple parameters in the same code block.

Unlike the *else statement*, you can have multiple *elif statements* in an *if-elif-else statement*. If the condition of the *if statement* is *False*, the condition of the next *elif statement* is checked. If the first *elif statement* condition is also *False*, the condition of the next *elif statement* is checked, etc. If all the *elif* conditions are *False*, the *else statement* and its indented block of statements is executed.

In Python, *if-elif-else statements* have the following syntax:

```
if condition1:

    indented Statement(s)

elif condition2:

    indented Statement(s)

elif condition3:

    indented Statement(s)

elif condition4:

    indented Statement(s)

else:

    indented Statement(s)
```

What happens if we want to test multiple conditions? This is where elif comes in.

```
num = int(input("Please input a number between 1 and 100: "))
if num < 12:
    print("The value you entered is lower than 12")
elif num > 13:
    print("The value you entered is higher than 13")
elif num < 13:
    print("The value you entered is lower than 13")
else:
    print("The value you entered is 13")
```

Try copying and pasting the code above into VS Code and running it with different numbers as input. Once you're happy, you can follow the logic, try changing the numbers used and creating different output strings until you really feel comfortable with how *if-elif-else* works.

Remember that you can combine *if*, *else*, and *elif* into one big statement. This is what we refer to as a *conditional statement*.

Some important points to note on the syntax of *if-elif-else statements*:

- Make sure that the *if-elif-else statements* end with a colon (the '**:**' symbol).
- Ensure that your indentation is done correctly (i.e., statements that are part of a certain control structure's 'code block' need the same indentation).
- To have an *elif*, you must have an *if* above it.
- To have an *else*, you must have an *if* or *elif* above it.
- You can't have an *else* without an *if* – think about it!
- You can have many *elif statements* under an *if*, but you may only have one *else* right at the bottom. It's like the fail-safe statement that executes if the other *if-elif* statements all evaluate to *False*!

Remember this overview of the basic structure you're learning.

SIMPLE EXAMPLES TO TRY

Take a look at the following example:

```
current_time = 12

if current_time < 11:

    print("Time for a short jog - let's go!")

else:

    print("It's after 11 - it's lunch time.")
```

If the condition of the *if statement* is *False* (**current_time** ends up being greater than **11**), the *else statement* will be executed. In this example, what do you think would happen if the time was set to 11 exactly? Copy, paste into VS Code, and run the code sample above. Do you think we should amend the code at all to handle this case? How would you do that? Try adding to the code sample and running it.

Another example of using an *else statement* with an *if statement* can be found below, but this one includes an *elif*. The value that the variable **hour** holds determines which string is assigned to the variable **greeting**:

```
hour = 18

if hour < 18:

    greeting = "Good day"

elif hour < 20:

    greeting = "Good evening"

else:

    greeting = "Good night"

print(greeting)
```

How could you change the code above to get the value of **hour** from the user, so that the output of the program would be dependent on the user's input? What would you have to do to the input value to use it in the logical tests of the conditions? Copy, paste, and run the code sample above first, and then try amending it to accept and act on user input. Hopefully, by now, you are starting to see the immense value that *if-elif-else* statements can offer to you as a programmer!

LET'S APPLY OPERATORS

Operators are symbols that tell the computer which mathematical calculations to perform or which comparisons to make.

We can combine the greater than, less than, and not operator with the equals operator and form three new operations.

- greater than or equal to `>=`
- less than or equal to `<=`
- not equal to `!=`

Comparing strings:

```
my_name = "Tom"

if my_name == "Tom":

    print("I was looking for you")
```

Comparing numbers:

```
num1 = 10

num2 = 20

if num1 >= num2:  # The symbol for 'greater than or equal to' is >=

    print("It's not possible that 10 is bigger than or equal to 20.")

elif num1 <= num2:  # The symbol for 'less than or equal to' is <=

    print("10 is less than or equal to 20.")

elif num1 != num2:  # The symbol for 'not equal to' is !=

      print("This is also true since 10 isn't equal to 20, but the elif statement before comes first and is true, so Python will execute that and never get to this one!")

elif num1 == num2:  # The symbol for 'equal to' is ==

    print("Will never execute this print statement...")
```

The program will check the first part of the *if statement* (is `num1` bigger than or equal to `num2`?).

If it is not, then it goes into the first *elif statement* and checks if `num1` is less than or

equal to `num2`. If it is not, then it goes into the next *elif statement*, etc. In the example above, the first *elif* will execute.

LOGICAL OPERATORS

A *logical operator* is another type of operator that is used to control the flow of a program. Logical operators are usually found as part of a control statement such as an *if statement*. Logical operators basically allow a program to make a decision based on multiple conditions; for instance, if you would like to test more than one condition in an *if statement*.

The three logical operations are:

Operator	**Explanation**
`and`	Both conditions need to be True for the whole statement to be *True* (also called the **conjunction operation**).
`or`	At least one condition needs to be *True* for the whole statement to be *True* (also called the **disjunction operation**).
`not`	The statement is *True* if the condition is *False* (only requires one condition).

The '`and`' and '`or`' operators require two operands, while the '`not`' operator requires one.

Let's take a real-life situation. When buying items at a store, two criteria need to be met. The item needs to be in stock and you need to have enough money to pay for it. This is an example of a *conjunction operation*, where both conditions need to be *True* for the whole statement to be *True*. This would be represented using the `and` operation.

Let's look at another situation. A person could receive a good mark at school either because they are very bright or because they studied hard. In this instance, either one of the options can be *True*, or both can be *True*, but at least one needs to be *True*. This is a *disjunction operation*, where at least one of the conditions needs to be met for the whole statement to be *True*. This would be represented using the `or` operation.

Example of an **and** operation:

```
team1_score = 3

team2_score = 2

game = "Over"

if (team1_score > team2_score) and (game == "Over"):

    print("Congratulations Team 1, you have won the match!")
```

Example of an **or** operation:

```
speed = int(input("How many kilometres per hour are you travelling at?"))

belt = input("Are you wearing a safety belt?")

if (speed > 80) or (belt != "Yes"):

    print("Sorry Sir, but I have to give you a traffic fine.")
```

ARITHMETIC OPERATORS

The arithmetic operators in Python are as follows:

Arithmetic operations	**Symbol used in Python**
Addition: adds values on either side of the operator.	**+**
Subtraction: subtracts the value on the right of the operator from the value on the left.	**–**
Multiplication: multiplies values on either side of the operator.	*****
Division: divides the value on the left of the operator by the value on the right.	**/**
Modulus: divides the value on the left of the operator by the value on the right and returns the remainder. E.g. `4 % 2 == 0` but `5 % 2 == 1`.	**%**

Exponent: performs an exponential calculation, i.e. calculates the answer of the value on the left to the power of the value on the right.	****
Assignment operations	**Symbol used in Python**
Equals: set the value of the thing on the left to that of the thing on the right. e.g. `n = 7`	**=**
Plus-equals: adds `1` to the variable for each iteration. e.g. `n += 1` is shorthand for `n = n + 1`. (This is particularly useful when using loops, as you will eventually see.)	**+=**
Minus-equals: minuses `1` from the variable for each iteration. e.g. `n -= 1` is shorthand for `n = n - 1`.	**-=**

Instructions

Before you get started, we strongly suggest you use an editor such as VS Code to open all text files (.txt) and Python files (.py).

First, read the accompanying Python example files. These examples should help you understand some simple Python. You may run the examples to see the output. Feel free to also write and run your own example code before doing the tasks to become more comfortable with Python.

Take note:

The tasks below are **auto-graded**. An auto-graded task still counts towards your progression and graduation.

Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete - you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task 1

Follow these steps:

- Create a new Python file for this task and call it **manipulation.py**.
- Ask the user to enter a sentence using the `input()` function. Save the user's response in a variable called `str_manip`.
- **Explore the string methods in here** to help you solve the problem below:
- Using this string value, write the code to do the following:
    - Calculate and display the length of `str_manip`.
    - Find the last letter in `str_manip` sentence. Replace every occurrence of this letter in `str_manip` with '@'.
        - e.g. if `str_manip` = "This is a bunch of words", the output would be: "Thi@ i@ a bunch of word@"
    - Print the last three characters in `str_manip` backwards.
        - e.g. if `str_manip` = "This is a bunch of words", the output would be: "sdr".
    - Create a five-letter word that is made up of the first three characters and the last two characters in `str_manip`.
        - e.g. if `str_manip` = "**Thi**s is a bunch of wor**ds**", the output would be: "Thids".

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded task 2

Follow these steps:

- Create a new Python file called **numbers.py**.
- Ask the user to enter three different integers.
- Then print out:
    - The sum of all the numbers
    - The first number minus the second number
    - The third number multiplied by the first number
    - The sum of all three numbers divided by the third number

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded task 3

Follow these steps:

- Create a new Python file in this folder called **award.py**.
- Design a program that determines the award a person competing in a triathlon will receive.
- Your program should read in user input for the times (in minutes) for all three events of a triathlon, namely swimming, cycling, and running, and then **calculate** and **display** the **total** time taken to complete the triathlon.
- The award a participant receives is based on the **total time** taken to complete the triathlon. The qualifying time for awards is 100 minutes. **Display the award** that the participant will receive based on the following criteria:

Qualifying criteria	Time range	Award
Within the qualifying time.	0–100 minutes	Honorary colours
Within five minutes of the qualifying time.	101–105 minutes	Honorary half colours
Within 10 minutes of the qualifying time.	106–110 minutes	Honorary scroll
More than 10 minutes off from the qualifying time.	111+ minutes	No award

Be sure to place files for submission inside your **task folder** and click "**Request review**" on your dashboard.

Rate us

Share your thoughts

Hyperion strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.