



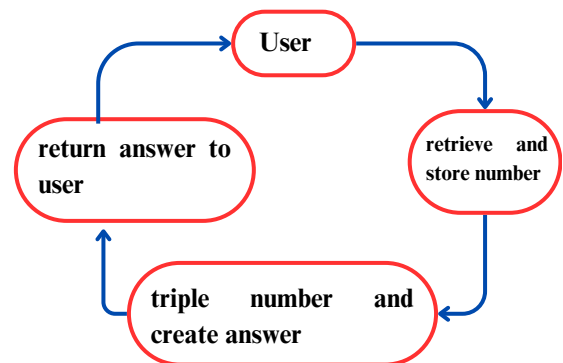
Introduction

Programs need to work with data. We can refer to 'data' for a program in a very simplified way as user inputs to the program, outputs the program must produce and give back to the user, internal values that we need to store for the program to work with, data from files, data housed in a database and so on. **Please Note** that 'data' and 'information' take on a more defined form when dealing with Statistics and Data Science as well as programming.

How do programs work with Data?

Consider a very simple program that would ask a user for a number, triple the value and then return the answer to the user

How does the program know it has a number?
Where do the input and output values go?



When it comes to dealing with data that a program stores and uses, we have to consider a few things:

- **The lifetime of the data** - is it only stored during the run of the program or do we need it stored beyond the program's execution
- **Level of access (scope)** - which 'parts' of the program can access and modify the data. Scope can also control the lifespan of our data
- **Datatype** - is the data a number, text, Boolean (True or False) or a complex type (coming later). The Datatype greatly impacts what can and cannot be done to the data

DataTypes

Datatypes do vary from language to language. In Python we will look at:

- Integers, Floats and Complex Numbers
- Booleans
- Strings (a char in other languages is a string in python with only one character)
- None
- Lists, Sets, Dictionaries and Tuples (**covered in note set 2**)

Before we jump into the different datatypes that Python gives us, we need to consider an important aspect of Python in that it is **Dynamically Typed** and briefly consider memory allocation for a variable.

Python uses an Interpreter which is responsible for doing many things, including determining the type that a variable has been given. **Dynamically Typed** means that we do **not** declare the type of our data

For example, if we want to create a variable called *number* and give it a value of *10*, we do not declare this to be an Integer. The Python interpreter will determine this on its own at runtime. It uses the value we have assigned to the variable to determine the datatype. There are other languages where this is not the case and we would have to declare the datatype when we declare the variable.

Variable Declaration and Instantiation are done at the same time in Python:

```
number = 10
```

Here I have **declared** the variable *number* and **instantiated** it with a value of 10

Memory

Consider that my above variable *number* needs to be stored in the computer's memory. To do so, the computer needs to **allocate** a certain amount of memory according to the datatype of the variable and in to which memory *section* the variable (or a reference to it's value) needs to go

Simply put, without assigning a value to the variable immediately the Interpreter has no information regarding required size and memory type that is needed and therefore cannot allow the variable to be stored in memory.

Variable Naming Rules

There are a few rules for correctly naming a variable. Some of the rules are compulsory and must be followed (failing to do so results in an Error and the program terminates). Other rules are conventions that should be followed (but would not result in an Error)

Rule	Valid	Invalid
Variable names can only start with a letter or underscore. However, variables that start with an underscore are usually for a different purpose	<pre>country = "Australia" total = 50 answer = 5.25 _coefficient = 3.259</pre>	<pre>3rd_attempt = True \$currency_value = 50</pre>
Variable names can only contain letters, underscores and numbers. No spaces are allowed in a variable name	<pre>daily_temp = 28 input_2 = 145 city_name = "Cape Town"</pre>	<pre>daily temp = 28 pass % = 95.58</pre>

Conventions:

Whilst a program can function without the following rules being followed, these rules are Python Conventions and should still be followed

- Python uses the **snake_case** convention. This means we start variable names with a lower case letter and then separate each word with an underscore. There is another convention followed in other languages called **Camel Case** which is shown below for comparison

snake_case

max_options = 6

camelCase

maxOptions = 6

- Variable names should describe what is being stored in the variable. Variable names such as 'number', 'a_word', 'value' and so on do not properly describe what value the variable is holding. In the valid examples above, the variables are more descriptive
- Variable names should be concise. This means we need to be descriptive with the variable name, but not make the name incredibly long. Generally we aim for a variable name with one or two words (three if really needed)
 - Good Variable Name: student_mark
 - Bad Variable Name: student_mark_from_school

Number Types

4

If we have a number that we want to work with, we have a few options:

Integers

With Integers we refer to numbers that have no decimal component such as 4 or 250000. Integers also include negative numbers as well such as -5.

The interpreter recognizes a variable as an Integer if it's value is a whole number that can be either positive, negative or 0 and contains no alphabet or special characters such as ';' or '['

```
employee_count = 50
account_balance = -450
car_cost = 275000
```

Python provides helpful functions for variable types

- `type(variable)`
returns the data type of a variable

```
employee_count = 50
datatype = type(employee_count)
print(datatype)
```



<class 'int'>

- `isinstance(variable, type to check for)`
checks if the variable type matches the type provided

```
employee_count = 50
type_check = isinstance(employee_count, int)
print(type_check)
```



True

We can also check for a match with multiple types by specifying the types in a **tuple** (a *complex datatype discussed in note set 2)

complex is a not reference to difficulty level, but rather specifies a datatype that can store other datatypes (or more of its own)

```
minute_value = 37
type_check = isinstance(minute_value, (int, float, str))
print(type_check)
```



True

Be Careful:

5

The decimal component of a number (even if it is zero) is how the interpreter recognizes a float

```
first_number = 100
second_number = 100.0
```

```
print(type(first_number))
print(type(second_number))
```



```
<class 'int'>
<class 'float'>
```

Floats

A float is a number that contains a decimal component

```
pass_percentage = 92.12
milli_conversion = 0.001
probability = 15.9
```

```
print(type(pass_percentage))
```



```
<class 'float'>
```

Floats can also make use of Scientific Notation:

It often happens that we may have a number with a large number of decimals. It can also occur that we have a number with a large number of whole number place values such as 1000000. With scientific notation we can make this number easier to represent.

Scientific notation uses multiplication by ten (with positive and negative powers) to shift the decimal point left or right. We are **not** changing the value of the number, only how it looks. With scientific notation from a Mathematics and Scientific perspective, we usually only want one number in front of the decimal point.

1. Number into Scientific Notation

Consider the number 15 million which is 15000000, we can shorten this using scientific notation. The *pretend* decimal point is currently at the end of the number so we want to move it forward or to the left (requiring a positive power for 10) and we need to move it 7 places.

15000000.0

$$15000000 = 1.5 \times 10^7$$

Alternatively, consider the number 0.000258

We can represent this number in scientific notation if we shift the decimal point to the right. To do this we have to use a negative power for ten. We will shift the decimal 4 places to the right. **Note** that the number of shifts equals the power (exponent) of ten

$$0.000258$$

Using Scientific Notation

$$0.000258 = 2.58 \times 10^{-4}$$

2. Scientific Notation to Number

To go from a number represented in Scientific Notation to one that is not, we have to use the above rules in a reverse manner.

- If the power of ten is **positive**, shift decimal to the **right**.
- **Negative** power, shift to the **left**

The shift count = power of ten

Consider: 1.42×10^3

Sometimes we add on a few trailing zeros to make the shifting easier if you are still getting used to it

$$1.420000 \times 10^3$$

$$1.42 \times 10^3 = 1420.0$$

Consider: 1.42×10^{-3}

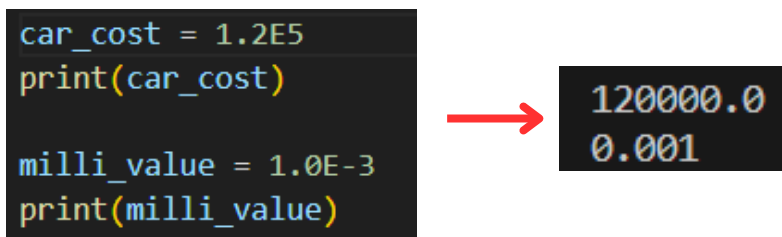
Here we will add a few leading zero's to make the next step easier

$$\begin{array}{c} 00001.42 \times 10^{-3} \\ \underbrace{\quad\quad\quad}_{3 \quad 2 \quad 1} \end{array}$$

$$1.42 \times 10^{-3} = 0.00142$$

Scientific Notation and Floats

In Python, we can also use Scientific Notation which is specified with an upper case or lower case 'E'. It does appear that the upper case is more common. In Python, the 'E' already means to the power of 10 so we just need to specify the exponent sign and value.



```
car_cost = 1.2E5
print(car_cost)

milli_value = 1.0E-3
print(milli_value)
```

→

```
120000.0
0.001
```

with:

$$1.2E5 = 1.2 \times 10^5$$

$$1.0E-3 = 1.0 \times 10^{-3}$$

Float size and Arithmetic

A float is a double precision value with a size of 64 bits. Floats are great for numbers with a small number of decimal places that can be accurately represented up to that number of decimal places

In programming, when we think of a number such as 8, this is not how it is stored in memory. Our computer will convert a number into its binary form.

There are some problems with this

Pretend we have a program in which we are performing a few arithmetic calculations:

```
first_float = 0.2 + 0.2
second_float = 0.2 + 0.2 + 0.2

print(first_float)
print(second_float)
```



0.4
0.6000000000000001

We were probably expecting the value **0.6** to be printed, but due to the ways in which floats are dealt with when storing them in memory, we can have issues such as this. We will look at rounding-off functions that can be used with floats which can assist us to some degree, but we should always consider this limitation with floats.

Rounding Off

Python allows us to round off our float values to a specific decimal amount which includes rounding to the nearest Whole Number (Integer).

Python already provides a built-in function called **round()** and from the math module we get **floor()**, **ceil()** and **trunc()**

Let's see them in action and then explain how they work

```
import math

rain_percentage = 85.6

round_value = round(rain_percentage)
trunc_value = math.trunc(rain_percentage)
ceil_value = math.ceil(rain_percentage)
floor_value = math.floor(rain_percentage)

print(round_value) 86
print(trunc_value) 85
print(ceil_value) 86
print(floor_value) 85
```

round()

round() performs rounding-off according to normal mathematics rules. If the digit concerned is greater than or equal to 5 then round up to the next value. If the digit is less than 5, then round down

round(number, decimal places)

We can also specify the number of decimal places we want to round off to, for no-decimal places (nearest whole number), set the argument value to None or leave it blank.

In the examples below, notice the rounding for positive and negative numbers

```
# Positive Numbers
print(round(1.14, 1))      # rounds to 1.1
print(round(1.16, 1))      # rounds to 1.2
print(round(300.489, 2))   # rounds to 300.49
print(round(300.489))      # rounds to 300

# Negative Numbers
print(round(-1.1))         # rounds to -1
print(round(-1.6))         # rounds to -2
```

Consider the data types too

```
number = 1.58

first_round = round(number)      # 2
second_round = round(number, 1)  # 1.6

print(type(first_round))         # <class 'int'>
print(type(second_round))        # <class 'float'>
```

math.trunc()

This function simply removes any decimal places from the number. It does not perform any rounding on the number

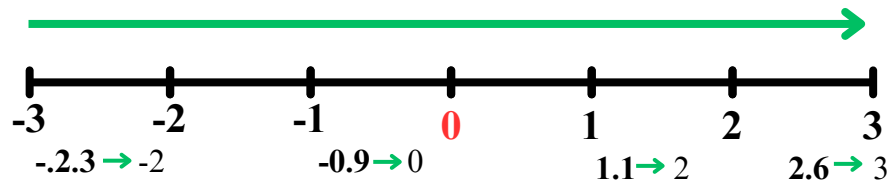
```
# Positive Numbers
print(math.trunc(1.16))      # 1
print(math.trunc(300.489))   # 300
print(math.trunc(300.99))    # 300

# Negative Numbers
print(math.trunc(-1.1))      # -1
print(math.trunc(-1.6))      # -1
```

By removing the decimal component of a number, **math.trunc()** will always return an integer value

This function will always round upwards towards positive infinity

- For **positive numbers**, this simply means it will round upwards
- For **negative numbers**, it means the same, but remember that upwards is towards 0



Negative Numbers

```
print(math.ceil(-0.9))      # 0
print(math.ceil(-2.3))      # -2
print(math.ceil(-199.02))   # -199
```

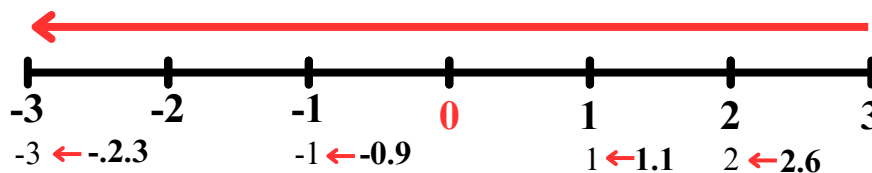
Positive Numbers

```
print(math.ceil(1.1))       # 2
print(math.ceil(2.6))       # 3
print(math.ceil(0.0001))    # 1
```

math.floor()

This function will always round downwards towards negative infinity

- For **positive numbers**, this simply means it will round downwards
- For **negative numbers**, it means the same, but remember that downwards is away from 0



Negative Numbers

```
print(math.floor(-0.9))     # -1
print(math.floor(-2.3))     # -3
print(math.floor(-199.02))  # -200
```

Positive Numbers


```
print(math.floor(1.1))      # 1
print(math.floor(2.6))      # 2
print(math.floor(0.0001))   # 0
```

Complex Numbers

11

Complex Numbers and their theory are not covered in these notes. For simple representation, these are numbers consisting of a Real Part and Imaginary Part. The Imaginary Part is denoted with the letter 'j'


```
complex_number = 2 + 3j  
print(type(complex_number))
```

 `<class 'complex'>`

Python Decimal Module - Basics

Earlier we looked at floats and noted problems that can occur with Floating Point Arithmetic seeing an example with unexpected decimals at the end of the number

```
first_float = 0.2 + 0.2  
second_float = 0.2 + 0.2 + 0.2  
  
print(first_float)  
print(second_float)
```

 `0.4`
`0.6000000000000001`

With floating point arithmetic, if we are only working with a very small number of decimal places (not interested in high levels of precision) then this can be managed and worked with. **However**, that extra 0.0000000000000001 that we see above can have a significant impact on calculations. If we consider Compound Interest with a person's money being worked with for decades (think of a pension fund) then this seemingly insignificant decimal can have an effect causing inaccuracies that continuously deviate answers further and further from their correct values.

The **Python Decimal Module** can help us. It is **not a datatype**, but it is recommended to become comfortable using it early on to ensure our calculations (requiring high precision) are accurate.



To use the decimal module it must first be imported. (Like the **math** module to use the **.trunc()**, **.ceil()**, and **.floor()** functions that we have seen)

```
from decimal import *
```

The example below will show the power of the **Decimal Module** with consideration for the type returned

```
from decimal import *

float_val = 0.2 + 0.2 + 0.2
print(f'Result from Float: {float_val}')
print(f'Type from float calculation: {type(float_val)}")

decimal_val = Decimal('0.2') + Decimal('0.2') + Decimal('0.2')
print(f'\nResult from Decimal: {decimal_val}')
print(f'Type from decimal calculation: {type(decimal_val)}")
```

The * means 'all'

This is called an **f-string** and is seen later in the **strings** section

```
Result from Float: 0.6000000000000001
Type from float calculation: <class 'float'>

Result from Decimal: 0.6
Type from decimal calculation: <class 'decimal.Decimal'>
```

- The first thing to notice is that the **decimal** module has now allowed us to perform an arithmetic calculation that has resulted in the value (0.6) as expected
- To create a 'decimal' (the correct way to say this is to create an Object of the Decimal Class), we use the **Decimal** Constructor. You can pass a string or float into the constructor, but a **string is recommended**. Passing a float can cause unexpected behaviour as shown below

```
from decimal import *

decimal_float = Decimal(0.2) + Decimal(0.2) + Decimal(0.2)
print(f'\nResult from Decimal with Floats: {decimal_float}')

decimal_string = Decimal("0.2") + Decimal("0.2") + Decimal("0.2")
print(f'\nResult from Decimal with Strings: {decimal_string}')
```

```
Result from Decimal with Floats: 0.6000000000000000333066907388
Result from Decimal with Strings: 0.6
```

A **constructor** is a special function that allows us to create a new instance (object) of a class. Classes are not covered in these notes, but simply think of them (specific to the Decimal module) as a means to create the decimal numbers that we want to work with.

The **types** in the above example showed us that we created an object of class: decimal.Decimal

The Boolean Datatype has only two possible values: **True** and **False**. Booleans are very important in testing results and in allowing a program to perform decision-making actions.

```
learning_programming = True
can_fly = False
```

To use Booleans effectively, we want to consider if-statements


if, elif and else statements - Overview

if-statements are essentially blocks of code that evaluate a specified condition. If the condition is met (True), then the code contained within the if-statement is executed. If not, it is skipped and program execution moves to the next line after the if-statement

Consider a program that prints a response based on the battery level of a laptop:

```
# define the battery level
battery_level = 70

if (battery_level > 50):
    print("Battery Saving Mode is not yet needed")
```



```
Battery Saving Mode is not yet needed
```

- The statement: **'battery_level > 50'** will check if the battery_level value is greater than 50 and we have set it to 70. This is correct and will return the Boolean value **True**.
- The if-statement has received **True** and will therefore execute the code within it's block which is to print the message seen above

What happens if the battery level was less than 50?

```
# define the battery level
battery_level = 30

if battery_level > 50:
    print("Battery Saving Mode is not yet needed")
```


In this situation, nothing happens (there is no output).

The condition **'battery_level > 50'** was not met so a Boolean **False** was given to the if-statement and the code within the if-statement did not execute

To make the program more useful, we will now incorporate an **else** statement. An else-statement means that if the if-statement should receive a False value, then move to the else statement and execute the code within it

```
# define the battery level
battery_level = 30

if battery_level > 50:
    print("Battery Saving Mode is not yet needed")
else:
    print("Battery Saving Mode has been turned on")
```



```
Battery Saving Mode has been turned on
```

We now have a more useful program. Notice how the use of Booleans (**True** and **False**) has given our program the ability to make a decision.

But there are times when we need to evaluate more than just two cases, such as evaluating a user's input for a desired menu option. For this we can use an **elif** (else-if statement). This statement **must** come after an if-statement or another elif-statement

The code below does not actually request or retrieve any input, but the **input()** function in Python can be used to perform this action. Here we focus mainly on Booleans and their application to if statements

```
menu_selection = "stock"


if menu_selection == "sale":
    print("Opening New Sale Window")

elif menu_selection == "manager":
    print("Opening Manager Dashboard")

elif menu_selection == "stock":
    print("Opening Stock Order Page")

elif menu_selection == "admin":
    print("Opening Admin Window")

else:
    print("Invalid Input has been recieved")
```



```
Opening Stock Order Page
```

The program will check through every if and elif-statement until a match is found. In the above example, the program stops at the line **menu_selection == "stock"** as the condition is met (**True**). Execution moves into the elif statement which prints the output "Opening Stock Order Page"

Any remaining elif and/or else statements are then skipped and execution will move to the next line after these elif and/or else-statements

There is more to if-statements as well as the use of Booleans, this is just one potential application

The String Datatype allows us to store and work with sequences of characters. Consider wanting to store the name of the language we are working with in a variable:

```
programming_language = "Python"
```

The Python interpreter can recognize a variable as a string if its value is surrounded by single, double or triple quotation marks.

```
field_of_study = 'Software'
programming_language = "Python"
year = """2024"""

print(type(field_of_study))
print(type(programming_language))
print(type(year))
```



```
<class 'str'>
<class 'str'>
<class 'str'>
```

Triple Quotation marks are better used in two instances:

- They allow us to create multi-line strings:

```
crazy_adventure = """I woke up on a cloud this morning. The cloud felt cold
and soft. I walked to edge and looked down at the Earth. Massive sections
of blue and green covered its surface. I couldn't see my house, or
the town I live in and this frightened me. I jumped from the edge and began to
fall through the cloud and I screamed.
Then I woke up.
"""
print(crazy_adventure)
```

```
I woke up on a cloud this morning. The cloud felt cold
and soft. I walked to edge and looked down at the Earth. Massive sections
of blue and green covered its surface. I couldn't see my house, or
the town I live in and this frightened me. I jumped from the edge and began to
fall through the cloud and I screamed.
Then I woke up.
```

Notice how everything that is typed is matched exactly in the output such as where the line ends and any white-space that is created in the string

- Triple Quotation marks are also used for docstrings. Docstrings are also multi-lined strings, but they explain the purpose and contents that a module, class and function has. Docstring formatting should match the rules and styles provided in the **PEP-8** guide available from the Python Website

Note: This note set does not focus on functions and the code below is only meant to show the docstring for a function to demonstrate its usage.

```
def is_even(number):
    """
    Determine if a number is even or odd.

    :param number: number to be checked
    :type number: int

    :returns: True or False
    :rtype: bool
    """
    if number % 2 == 0:
        return True
    else:
        return False

print(is_even(4))
print(is_even(3))
```

→ True
False

Quotation mark combinations

We can use single, double and triple quotation marks with our values to initiate a string, **but** we need to be careful when using combinations. Consider declaring and initializing a string below

```
plant_description = "The Jacaranda Mimosifolia Bonsai or Blue Jacaranda is a popular plant"
```

The Interpreter recognizes the string by matching a starting double quotation mark (in this case) with one at the end of the string. Now we consider:

```
plant_description = "The Jacaranda Mimosifolia Bonsai or "Blue Jacaranda" is a popular plant"
```

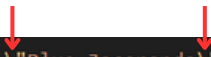
The Interpreter is unable to recognize the remainder of the string as we have used multiple double quotation marks, but this can be solved by using single and double quotation marks:

```
plant_description = "The Jacaranda Mimosifolia Bonsai or 'Blue Jacaranda' is a popular plant"
```


In Python Strings, we can use a backslash '/' to **escape** characters. These are characters that have special meaning. Above we saw that there can be problems in a string if we have multiple sets of double quotation marks.

Special Characters can help:

```
plant_description = "The Jacaranda Mimosifolia Bonsai or \"Blue Jacaranda\" is a popular plant"
print(plant_description)
```



Notice that the backslashes are not printed in the output

```
The Jacaranda Mimosifolia Bonsai or "Blue Jacaranda" is a popular plant
```

We have 'escaped' the inner, double-quotation marks meaning the interpreter understands these inner quotation marks are not part of the string declaration and are meant to be treated only as quotation marks

Table of Common Escape Characters


Name	Code	Example
Single Quotation	\'	<pre>sentence = 'The sun\'s rays are bright' print(sentence)</pre> The sun's rays are bright
Double Quotation	\"	<pre>reported_speech = "The instructor said: \"Great Job!\"" print(reported_speech)</pre> The instructor said: "Great Job!"
Tab	\t	<pre>columns = "Title:\tGenre:\tPrice:" print(columns)</pre> Title: Genre: Price:
New Line	\n	<pre>rows = "Title:\nGenre:\nPrice:" print(rows)</pre> Title: Genre: Price:

Backslashes

A backslash is used to indicate an escape character in a string, but this means that if we want to use a backslash then we have to escape it too:

Consider wanting to state a file path for example without escaping the backslash:


```
file_path = "C:\Users\Admin\Desktop\CodeWork"  
print(file_path)
```



SyntaxError:

Escaping the backslashes:

```
file_path = "C:\\Users\\Admin\\Desktop\\CodeWork"  
print(file_path)
```



C:\Users\Admin\Desktop\CodeWork

NOTE:

The above example only demonstrates one real use of the backslash which can be in stating a windows file path as a string, but this is not a solution that will work for every OS or application that requires a filepath

Working with Strings - String Functions

Strings are incredibly powerful and control communication aspects of every program. Strings allow our applications to communicate information to our users and is often the datatype in which the user will communicate information back to the program.

Python provides us with many useful functions for String manipulation allowing us to work with strings. Some of these functions are given on the following pages.

Name	Code	Explanation and Example
len()	len(string)	Returns the length of a string <pre>item = "couch" characters = len(item) print(characters)</pre> 5
count()	string.count(substring)	Returns the frequency a substring appears in a string <pre>words = "cat bat mat cats green" cat_count = words.count("cat") print(cat_count)</pre> 2
find()	string.find(substring)	Returns index position of first occurrence of match. Returns -1 if no match is found <pre>01234 words = "cat bat mat cats green" bat_start_index = words.find("bat") print(bat_start_index)</pre> 4
index()	string.index(substring)	Returns index position of first occurrence of match. Returns ValueError if not found <pre>01234 words = "cat bat mat cats green" bat_start_index = words.index("bat") print(bat_start_index)</pre> 4
replace()	string.replace(old, new)	Replace a specified character with a new character. Does NOT modify the original string <pre>words = "cat bat mat cats green" replaced = words.replace("cat", 'dog') print(replaced)</pre> dog bat mat dogs green
split()	string.split(character)	Split a string into a list. Perform split at specified character. Note that the specified character is not retained <pre>scores = "1,5,3,2,1,8,10,4" score_list = scores.split(",") print(score_list)</pre> ['1', '5', '3', '2', '1', '8', '10', '4']

Name	Code	Explanation and Example
upper()	string.upper()	<p>Convert all alphabet characters to upper case. Does NOT modify original string</p> <pre>greeting = "Hello" new_greeting = greeting.upper() print(greeting) Hello print(new_greeting) HELLO</pre>
lower()	string.lower()	<p>Convert all alphabet characters to lower case. Does NOT modify original string</p> <pre>greeting = "Hello" new_greeting = greeting.lower() print(greeting) Hello print(new_greeting) hello</pre>
capitalize()	string.capitalize()	<p>Convert starting letter of first word to upper case, all others are converted to lower case. Does NOT modify the original string</p> <pre>description = "THE Best coding LaNguAge" description_corrected = description.capitalize() print(description_corrected)</pre> <p>The best coding language</p>
title()	string.title()	<p>Convert starting letter of each word to upper case and will correct the rest of the word to lower case. Does NOT modify the original string</p> <pre>description = "THE Best coding LaNguAge" description_corrected = description.title() print(description_corrected)</pre> <p>The Best Coding Language</p>
swapcase()	string.swapcase()	<p>Perform swap of lower and upper case. Does NOT modify the original string</p> <pre>description = "THE best CoDiNg language" description_corrected = description.swapcase() print(description_corrected)</pre> <p>the BEST cOdInG LANGUAGE</p>

Name	Code	Explanation and Example
startswith()	string.startswith(<i>substring</i>)	<p>Returns True if string starts with a specified substring. False if not</p> <pre> title = "Day in the Life" print(title.startswith("The")) print(title.startswith("day")) print(title.startswith('Day')) </pre> <p>False False True</p>
endswith()	string.endswith(<i>substring</i>)	<p>Returns True if string ends with a specified substring. False if not</p> <pre> title = "Day in the Life." print(title.endswith("Life")) print(title.endswith("Life.")) print(title.endswith('life')) </pre> <p>False True False</p>
isalpha()	string.isalpha()	<p>Returns True if string contains only alphabet characters. False if not</p> <pre> title_one = "3D printing" title_two = "Three_D_Printing" title_three = "ThreeDPrinting" print(title_one.isalpha()) print(title_two.isalpha()) print(title_three.isalpha()) </pre> <p>False False True</p>
isdecimal()	string.isdecimal()	<p>Returns True if string contains only numerical characters. False if not</p> <pre> numbers_one = "1234" numbers_two = "1234" print(numbers_one.isdecimal()) print(numbers_two.isdecimal()) </pre> <p>False True</p>
isalnum()	string.isalnum()	<p>Returns True if a string contains only alphanumeric (alphabet and numerical) characters. False if not</p> <pre> title_one = "5_hours_a_day" title_two = "5HoursADay" print(title_one.isalnum()) print(title_two.isalnum()) </pre> <p>False True</p>

Name	Code	Explanation and Example
isupper()	string.isupper()	<p>Returns True if all alphabet characters are in upper case. False if not</p> <pre>message_one = "GOOD MORNING" message_two = "Good Morning" print(message_one.isupper()) print(message_two.isupper())</pre> <p>True False</p>
islower()	string.islower()	<p>Returns True if all alphabet characters are in lower case. False if not</p> <pre>message_one = "good morning" message_two = "Good Morning" print(message_one.islower()) print(message_two.islower())</pre> <p>True False</p>
strip()	string.strip(<i>character</i>)	<p>Removes a specified character from the front (leading) and end (trailing) portion of a string. Does not modify the original string</p> <pre>search_input = "-Find-Location-" input_corrected = search_input.strip("-") print(input_corrected)</pre> <p>Find-Location</p> <p>If <i>strip()</i> argument is left blank, it will remove trailing and leading whitespace</p> <pre>search_input = " Option One " input_corrected = search_input.strip() print(input_corrected)</pre> <p>Option One</p>

'Chaining' string functions

23

Often we need to format strings during the run of an application. Consider the example below in which we retrieve input from a user for a simple menu selection component of an application.

```
menu_selection = input("Please enter the name of an option: ")
```

We want to compare the user's input against menu specified options.

```
def print_menu():  
    print("Menu:\n")  
    print("\tHome\n\tSongs\n\tPlaylists\n\tAbout\n\tExit")
```

Place menu print into a function allowing for repeated calls

Menu:

```
    Home  
    Songs  
    Playlists  
    About  
    Exit
```

When comparing the user's input against these options, we need to account for possible errors in the user's input such as trailing or leading whitespace and incorrect upper or lower case in addition to checking that an input was received in the first place:

```
def print_menu():  
    print("Menu:\n")  
    print("\tHome\n\tSongs\n\tPlaylists\n\tAbout\n\tExit")  
  
menu_selection = ""  
  
while menu_selection != "exit":  
    print_menu()  
  
    menu_selection = input("\nPlease enter the name of an option: ")  
  
    # check input was received  
    if menu_selection != "":  
        # remove leading and trailing whitespace  
        menu_selection = menu_selection.strip()  
        # convert input to lower case (removes case-sensitivity)  
        menu_selection = menu_selection.lower()  
  
        # print input for testing  
        print(f"Testing of Menu Selection: {menu_selection}")
```

```

Please enter the name of an option: About
Testing of Menu Selection: about
Menu:
    Home
    Songs
    Playlists
    About
    Exit

Please enter the name of an option: 

```

- Leading whitespace
- Notice that the input received started with a capital letter, but has been converted to lower case

Considering that `strip()` and `lower()` return new strings, we can **chain** our calls of these functions:

```

def print_menu():
    print("Menu:\n")
    print("\tHome\n\tSongs\n\tPlaylists\n\tAbout\n\tExit")

menu_selection = ""

while menu_selection != "exit":
    print_menu()

    menu_selection = input("\nPlease enter the name of an option: ")

    # check input was received
    if menu_selection != "":
        # remove leading and trailing whitespace and case-sensitivity
        menu_selection = menu_selection.strip().lower()

        # print input for testing
        print(f"Testing of Menu Selection: {menu_selection}")

```

Note:

The menu is not fully-functional and needs additional code to compare the user's input against the possible menu options from which a desired action is then executed

String formatting and String Immutability

In the above example (and earlier in the note set), you may have noticed the usage of an **f-string** such as:

```
print(f"Testing of Menu Selection: {menu_selection}")
```

String formatting allows us insert contents or ***modify** a string, but we must discuss modification of strings:

In Python (and other languages) strings are **immutable**. Immutable means that they cannot be changed after they have been created. When we modify a string, **the original string is actually destroyed** and a new one is created which is then assigned to the original variable name. The words **modification** and **formatting** are still used, but we do keep in mind that - behind the scenes - a new string is created.

Often we want to add content to a string and we can do this in a few ways

Concatenation

String Concatenation allows us to combine two or more strings using the “+” operator.

```
string_one = "Python"
string_two = " Programming"

programming_language = string_one + string_two
print(programming_language)
```

```
Python Programming
```

Keep in mind that this method will only allow a combination of strings and not other types such as an Integer and a String

```
currency = "R"
amount = 100

balance = currency + amount
print(balance)
```

```
TypeError: can only concatenate str (not "int") to str
```

One way to correct this and match string concatenation rules is to use a type-cast function. Here we will first convert the Integer ‘amount’ to a string and then perform concatenation

```
currency = "R"
amount = 100

balance = currency + str(amount)
print(balance)
```

```
R100
```

NOTE: The type-cast was not assigned to the variable ‘amount’. This means that ‘amount’ is still an Integer.

```
currency = "R"
amount = 100

balance = currency + str(amount)
print(balance)
print(type(amount))
```

```
R100
<class 'int'>
```

Apart from adding (concatenating) completed strings together, we can also add contents into specified portions of a string

format() method

The format() method allows us to add specified values into a string using

- non-specified ordering
- specified ordering
- naming

The examples below will show each case:

Non-specified Ordering

Here the order of the items to add into the string is followed

```
income = 500
expenses = 200
profit = income - expenses

finance_statement = "Income = {}\nExpenses = {}\nProfit = {}".format(income, expenses, profit)
print(finance_statement)
```

```
Income = 500
Expenses = 200
Profit = 300
```

Specified Ordering

Here we use numbering to specify the order in which we want the values to be inserted into the string

```
income = 500
expenses = 200
profit = income - expenses

finance_statement = "Income = {0}\nExpenses = {1}\nProfit = {2}".format(income, expenses, profit)
print(finance_statement)
```

```
Income = 500
Expenses = 200
Profit = 300
```

As long as we know the order in which we want the values to appear (and specify this), then the arguments of the `format()` method need only to match the specified order

```
income = 500
expenses = 200
profit = income - expenses

finance_statement = "Income = {1}\nExpenses = {2}\nProfit = {0}".format(profit, income, expenses)
print(finance_statement)
```

```
Income = 500
Expenses = 200
Profit = 300
```

Naming

Here we use variable names to insert values into a string

```
finance_statement = ("Income = {income}\nExpenses = {expenses}\nProfit = {profit}"
                    .format(income = 500, expenses =200, profit = 500))
print(finance_statement)
```

```
Income = 500
Expenses = 200
Profit = 500
```

The use of variable names removes the need for ordering of our values and allows us to use a value repeatedly with ease (such as *currency* below):

```
finance_statement = (
    ""Income = {currency}{income}
    Expenses = {currency}{expenses}
    Profit = {currency}{profit}""
    .format(income = 500, expenses =200, profit = 500, currency = "$"))
print(finance_statement)
```

```
Income = $500
Expenses = $200
Profit = $500
```

f-string

With an **f-string**, we can specify variable names directly within the string itself. We must append an **f** before the quotation marks.

```
currency = "$"
income = 500
expenses = 200
profit = income - expenses

finance_statement = f"Income = {currency}{income}\nExpenses = {currency}{expenses}\nProfit = {currency}{profit}"

print(finance_statement)
```

```
Income = $500
Expenses = $200
Profit = $300
```

None

Introduction:

In programming (and Mathematics) we are often faced with the difference between

- 0
- Does not Exist
- Undefined

We will briefly explore these ideas before working with Null

Consider having a new bank account with a balance of 0. The bank account itself does exist and has a value of 0. This is the idea of the **value of 0**.

Now consider the moment before the account was opened. We cannot say that the value of the account is nothing or 0 because the account does not exist. We have to instead say that the account does not exist. This is the idea of does not exist or **None** in programming

Undefined can be trickier, but is seen in a division by zero for example. **Undefined is not the same as None or Null**, as undefined (by its own name) means there is no definition, understanding or explanation for an operation or concept. A division by zero is undefined because there is no consistent logic that we have to explain what it means to divide by zero.

Extra: It is not uncommon to need to revisit or understand these differences. For those familiar with parabolas in Mathematics, we can use the *discriminant* value from a parabola to determine if it has roots (x-intercepts). In the event that it does not, attempts to factorize the parabola (even with the use of the quadratic formula) would result in a non-real number. This does not mean that the parabola does not exist or that it can't be worked with - but rather that it simply does not intercept or touch the x-axis.

None in programming

In programming, None (or null) means that a value does not exist for a particular variable. It is a clever data type that allows to work with (and perform checks) using a variable that has no value

None is not a number. It does not mean False in the case of Booleans. It does not mean an empty string.

Consider the following example in which we think of two functions (not discussed in these notes). We want the first function to perform an operation and print the result, the second function will perform the same operation but return the result to its calling method

```
def double_and_print(value):
    """Double and print result"""

    double_value = value * 2
    print(f"{value} doubled is {double_value}")

def double_and_return(value):
    """Double value and return result"""
    double_value = value * 2
    return double_value
```

The first function does not return anything, the second returns the doubled value. We will call these functions:

```
# first function performs print of double value
double_and_print(5)

# second function returns doubled value, we will print the results here
print(double_and_return(5))
```

```
5 doubled is 10
10
```

BUT. what happens if we attempt to retrieve a value from the first function (the one that prints the value for us)? Remember that we did not specify a return statement in this function

```
#attempt to retrieve a value from function that has no return statement
double_result = double_and_print(5)
print(f"return from function is {double_result}")
```

```
5 doubled is 10
return from function is None
```

The function will return nothing and hence we see that the return is **None**. There are many other occasions where the use of the **None** type can be very useful. The main thing to understand is that **None** allows us safely specify that a value does not exist for a variable.