3. DATA TYPES

The data types you will be using are predefined in the Python programming language. Choosing the correct data type to use is important because it determines what actions we can perform on the data, e.g. we cannot divide a text string by an integer. Every value you store will have one of the following data types:

STRING

- A string is a series of characters, in quotes. These can be single or double quotes.
- We can add them together, e.g. “birth” + “day” would give you “birthday”.
- Used to represent pseudo-numbers, e.g. a telephone number.

INTEGER

- An Integer is a whole number; it can be positive or negative.
- We can use a range of mathematical operators (+ - * / etc.) on integers.

REAL

- This enables us to store a number with a fractional part.
- A real number is sometimes called a float or single or double.

BOOLEAN

- A Boolean value can be True or False.
- Many questions have the answer ‘Yes’ or ‘No’.
- We want to find out whether things are true or not in our programs.

I can check what data type a variable is by using: *type (variable_name)*.

```
>>> sunny = True
>>> type(sunny)
<class 'bool'>
>>> weather = "cloudy"
>>> type(weather)
<class 'str'>
```

Why is this important?

You might use this to ensure that the result of a calculation is shown as a decimal as this may affect the rest of your program and give a false result.

You may want to compare values which are not the same data type, e.g. your program takes input as a string and then needs to be converted to an integer to check whether the value is greater than or less than a numeric value.

Look at this example, which causes a Type Error when the code is executed in the interpreter:

```
>>> legal_age = 17
>>> student_age = input("Enter your age: ")
Enter your age: 18
>>> if student_age >= legal_age:
        print("You are old enough to drive")
else:
        print("You cannot drive yet, you are too young")


Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    if student_age >= legal_age:
TypeError: unorderable types: str() >= int()
>>>
```


3.1 CASTING DATA TYPES

Python assumes that all data entered using the **input()** built-function is a STRING unless the data type is CAST into a different data type. This means telling Python to treat the data input as an INTEGER or a FLOAT.

In the improved version, the data input is CAST to an INTEGER when the input is requested. This could also be done like this:

```
>>> student_age = input("Enter your age: ")
Enter your age: 18
>>> student_age = int(student_age)
>>> type(student_age)
<class 'int'>
```

Used to show the data type has changed.

The example below is more **efficient** as only one line of code has been used to cast the data type rather than two lines of code as shown above.

Efficient Code:

```
>>> legal_age = 17
>>> student_age = int(input("Enter your age: "))
Enter your age: 18
>>> if student_age >= legal_age:
        print("You are old enough to drive")
else:
        print("You cannot drive yet, you are too young")


You are old enough to drive
```

More examples of casting data types:

```
>>> student_age = '16'
>>> type(student_age)
<class 'str'>
>>> type(int(student_age))
<class 'int'>
>>> type(float(student_age))
<class 'float'>
>>> type(str(student_age))
<class 'str'>
```


3.2 GETTING DATA INPUT

We have already seen examples of the built-in function input() being used to get data from a user. The efficient code example above shows how the input from the user can be CAST into the correct data type by putting the input() code inside a data type.

```
>>> name = input("Please enter your name: ")
Please enter your name: Adam
>>> age = int(input("Please enter your age: "))
Please enter your age: 15
>>> print("Hello",name,",you are",age,"years old")
Hello Adam ,you are 15 years old
```

3.3 STRINGS

As already mentioned in Data Types, strings are sequences of characters enclosed by speech marks. They can be enclosed by single quotes, double quotes or three quotes of either type. As you can see, the output from each version is the same.

```
#These are all the same

my_string ='Hello world'
print(my_string)
my_string = "Hello world"
print(my_string)
my_string = '''Hello world'''
print(my_string)
my_string = """Hello world"""
print(my_string)

#triple quotes allow the string to cover several lines

my_string = """
    The string "Hello world" is the
    first text that people learning
    to code output to screen"""
print(my_string)
```

```
>>>
Hello world
Hello world
Hello world
Hello world

    The string "Hello world" is the
    first text that people learning
    to code output to screen
```

Which one should I use?

- Using single quotes means that you need to use escape characters in the string if you also want to use characters like a backslash, an apostrophe or double quotes.
- Using double quotes means that you do not need to use escape characters.

Using triple quotes means that text can span several lines, although these are generally used for docstrings in functions; see Docstring and Comments for more information.

Escape Characters

Some useful escape characters are shown below.

Escape Character	What it does
\\	Allows the use of a backslash inside a single quotes string
\'	Allows the use of an apostrophe inside a single quotes string
\"	Allows the use of double quotes inside a single quotes string
\n	ASCII linefeed-newline
\t	Horizontal tab (indents your text string)

Examples:

```
>>> print("The quick brown \nfox jumped over\nthe lazy dog")
The quick brown
fox jumped over
the lazy dog
>>>
>>> print("Please choose from\n\t1)Play Game\n\t2)Quit")
Please choose from
        1)Play Game
        2)Quit
```

```
>>> print('Don't open that door!')

SyntaxError: invalid syntax
>>> print('Don\'t open that door!')
Don't open that door!
>>> print('This is a backslash \')

SyntaxError: EOL while scanning string literal
>>> print('This is a backslash \\')
This is a backslash \
>>> print("Everyone says "hello" ")
SyntaxError: invalid syntax
>>> print("Everyone says \"hello\"")
Everyone says "hello"
>>>
```


STRING OPERATIONS

Strings are immutable; this means that once we have created a string variable we cannot alter it or edit it.

There are a number of string operations we can perform on a string variable that will be useful when writing your code. Here are some common examples:

- *len (myString)* – returns the number of characters in the string, including spaces.
- *myString.upper()* – returns the string in upper case.
- *myString.lower()* – returns the string in lower case.
- *myString.capitalize()* – returns the string with the first letter of the string capitalised.
- *myString.title()* – returns the string with the first letter of each word capitalised.
- *myString.replace(x, y)* – returns the string with the characters represented by x replaced by the characters represented by y.
- *myString [x: y]* – returns the a substring of the original string starting at character x and ending before character y.

Examples of use:

```
>>> my_string = 'Homer Simpson'
>>> len(my_string)
13
>>> my_char = my_string[6]
>>> my_char
'S'
>>> print(my_string[0:5])
Homer
>>> print(my_string.upper())
HOMER SIMPSON
>>> title = 'this is my title'
>>> print(title.capitalize())
This is my title
>>> print(title.replace('i','z'))
thzs zs my tztle
>>>
```

Why is this important?

Look at this example; I am trying to test whether the two strings are the same by using the equality operator.

```
>>> text_1 = "Paris"
>>> text_2 = "PARIS"
>>> text_1 == text_2
False
>>> text_1.upper() == text_2
True
>>>
```

If I test equality WITHOUT converting the first string to upper case, the result of the comparison is false; Python does not recognise that the strings are the same.

FORMATTING STRINGS

When we need to use variables inside a print statement there are a number of different ways we can do this:

```
>>> teacher = input("Please input your teacher's name: ")
Please input your teacher's name: Mr Jones
>>> print("Hello ",teacher)
Hello  Mr Jones
>>> print("Hello " + teacher)
Hello Mr Jones
>>> print("Hello {0}".format(teacher))
Hello Mr Jones
>>>
```

1. Use a comma to add the variable into the print statement.
2. Concatenate the string with the variable inside the print statement.
3. Use the string method .format().

The last method may seem more time-consuming than the first two, but it allows a lot more flexibility.

Example:

```
>>> euro = 1.39
>>> cash = 250.0
>>> print("£{0} will buy {1} Euros at today's rate: {2}".format(cash, euro * cash, euro))
£250.0 will buy 347.5 Euros at today's rate: 1.39
>>>
```

String alignment tricks:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>>
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>>
>>> '{:^30}'.format('centred')
'           centred             '
>>>
>>> '{:*^30}'.format('centred')
'***********centred************'
>>>
```

Character	Alignment
<	left alignment
^	centre alignment
>	right alignment

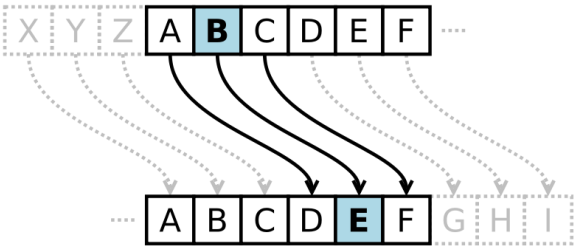
We can also use additional 'control characters' to improve the way any variable types are printed.

The code in this screenshot has been created in script mode. This means it can be saved and run by pressing F5; the filename must have the file extension .py.

```
#student test results

s1 = "Max"
s2 = "Chloe"
s3 = "Tom"
s4 = "Emily"

test1 = 60
test2 = 74
test3 = 72
test4 = 59

print("Student  Test ")
print("{0} {1}".format(s1, test1))
print("{0} {1}".format(s2, test2))
print("{0} {1}".format(s3, test3))
print("{0} {1}".format(s4, test4))
print("\n\nImproved version:")

# example {0:<10} means left aligned, 10 spaces available for the string
print("{0:<12} {1:<5}".format("Student ","Test "))
print("{0:<12} {1:<5}".format(s1,test1))
print("{0:<12} {1:<5}".format(s2,test2))
print("{0:<12} {1:<5}".format(s3,test3))
print("{0:<12} {1:<5}".format(s4,test4))
```

```
Student  Test
Max 60
Chloe 74
Tom 72
Emily 59


Improved version:
Student      Test
Max          60
Chloe        74
Tom          72
Emily        59
>>>
```

USING ORD() AND CHR()



These two built-in functions are commonly used when creating a simple Caesar Cipher program to encrypt a text string.

A Caesar Cipher simply substitutes the actual character in the string with another letter a certain number of spaces further on in the alphabet.

The ord() function allows us to represent each letter as an ordinal number. We can then add the required number of letters to shift by, which must be between 1 and 26, to find the value of the new number. The new number can be changed back to a letter by using the chr() function.

The numbers used to represent each upper or lower case letter in the alphabet are based on the ASCII character set. Computers only understand binary, so the ASCII codes represent text and other punctuation symbols. The diagram shows some of the letters and their numerical representations in binary, octal, decimal and hexadecimal.[1]

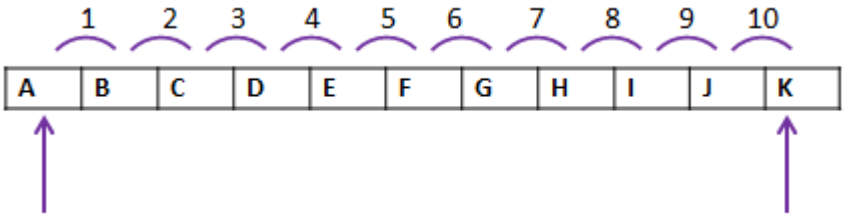
Binary	Oct	Dec	Hex
100 0000	100	64	40
100 0001	101	65	41
100 0010	102	66	42
100 0011	103	67	43

Binary	Oct	Dec	Hex
110 0000	140	96	60
110 0001	141	97	61
110 0010	142	98	62
110 0011	143	99	63

[1] Binary is Base 2, Octal is Base 8, Decimal is Base 10, Hexadecimal is Base 16

Example:

```
>>> ord('A')
65
>>> ord('a')
97
>>> letter = 'A'
>>> new_letter = ord(letter) + 10
>>> chr(new_letter)
'K'
>>>
```



This simple example shows how we can use these two built-in functions to write a basic Caesar Cipher. The key is 7 and **z** is 7 letters away from **s** in the alphabet.

```
password = "secret"
encrypted = ''    # empty string for the encrypted password

# encrypt the password
key = 7                # the number of letters to the right
for each in password:
    new_letter = ord(each)+key
    encrypted += chr(new_letter) # each new letter is added to the encrypted word

print(encrypted)
```

```
zljyl{
>>>
```

EXERCISE 03: STRINGS

Complete these exercises in script mode in IDLE.

1. Prompt the user to enter their name. Output a message: 'Hello (name), I hope you are well'; the message should be output on two lines using one print statement.
2. Prompt the user to enter two whole numbers. Output a message 'The average of your two numbers, (number 1) and (number 2) is (average)'.
3. Prompt the user to enter a string for encryption and a key value between 1 and 26. Output the original string and the encrypted string with a suitable message.

