Lesson 3
Assignment and Variables

Giving a variable a value is called **assignment**. We can use assignment to give a value to a variable in a program without input. For example, suppose we have a variable called `metres`, to which we wish to give the value `12`. In Python we write:

```
metres = 12
```

This is usually read as '**`metres` is assigned the value 12'**. We can use any value instead of `12`. Other examples of assigning values to variables are:

```
centimetres = 50
litres = 10.5
metres = 4
colour = 'red'
name = 'Joe Carthy'
pay_per_hour = 11.5
```

.

Example L3.1: Write a program to convert 5 metres to centimetres. A simple Python program to do this is given below.

```
#convert1.py: converts metres to centimetres
#Author: Joe Carthy
#Date: 21/10/2023

metres = 5
centimetres = metres * 100
print('5 m is ', centimetres, 'cms',)
```

Executing this program produces as output:

```
5 m is 500 cms
```

Here we use the value of the variable `metres` to compute the value of the variable `centimetres`.

.

We can assign a variable a value using other variables or by specifying the value directly by specifying a number or a string.

```
gallons = 4
pints = gallons * 8
kilos = 4
metres = 18
cms = (kilos * 100000) + (metres * 100)
name = 'Joe'
```

In programs, values such as

```
4, 10.5, 'joe', name, 2 + 3, cms, gallons * 8
```

are called **expressions**

Arithmetic operators in Python

Python Operator	Operation
+	Addition
−	Subtraction
*	Multiplication
/	[Floating-point] Division
**	Power

Arithmetic expressions

```
pints = gallons * 8
cms = metres / 100
radius = 4
area_of_circle = 3.14 * radius **2
```

The integer numbers (eg 1, 2, 400, 200000) have type int

Numbers with a fractional part (eg 1.5, 2.444, 20.0) have type float

Undefined variable name errors are often caused by misspelling a variable name e.g.

```
metres = 25

print ('metres = ', metrs)
```

The variable `metrs` has not been defined

Variables must be defined before you use them

If a variable is ‘not defined” (not assigned a value), trying to use it will generate an error.

So if you run the 1 line program:

```
print ('x = ', x)
```

You get an error because `x` has not been defined with an error message such as: that below – the last line is the helpful one:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined
```

Example L3.2: Converting metres to centimetres, version 2.

```
#convert2.py: converts metres to centimetres version 2
#Author: Joe Carthy
#Date: 21/10/2023

m = input('Enter number of metres: ')

metres = float ( m )

centimetres = metres * 100

print(metres, 'metres is ', centimetres, 'cms',)
```

Executing this program produces as output:

```
Enter number of metres: 4
4.0 metres is 400.0 cms
```

.

Variable Types

The `input` function reads from the keyboard and returns a **list of characters** i.e. a string.

Thus the variable **m** in the example above contains the string '4' and **not** the number 4.

This is very confusing for beginners to programming.

A fundamental aspect of variables is that they have a **type**. The type of a variable tells you **what kind of data** it stores.

In our programs we will use three types: **int** (whole numbers), **float** (numbers with decimal point) and **string** (list of characters).

When you are working with numbers and wish to do arithmetic with them (add, subtract, multiply and divide) then you must use either the type **int** or **float**.

So it is crucial to understand the difference between the number 42 and the string '42' as used in the following code:

```
a = 42
b = a * 2
```

This results in b having the value 84.

The result is the integer number 84

In order to arithmetic with variables, the **variables must be** of type **`int`** or type **`float`**

Now consider the code

```
x = '42'     # x is type string
y = x * 2    # y is type string
```

This results in `y` having the value `4242` – a string of characters.

When you ***'multiply'*** a string variable by a number `n` you get `n` copies of the string e.g. the code:

```
x = 'bye'
y = x * 3
```

gives `y` the string value `'byebyebye'`

You cannot do numeric calculations with a string even when the string contains a number. This brings us back to Example L3.2 and the statements

```
m = input('Enter number of metres: ')

metres = float ( m )
```

The variable `m` is of type string.

The `float` function converts the string `m` to a number with a decimal point (also called a **floating point** or **real** number).

This means that `metres` now contains a number and we can do arithmetic with it.

Example L3.2 revisited.

The output of L3.2 is *'crowded'* in that there is no blank line before or after the output or between the two lines of output.

```
Enter number of metres: 4
4.0 metres is 400.0 cms
```

This makes it hard to read the output. You can use the `'\n'` character in strings to start new lines.

The version below addresses this issue by putting one '\n' in the `input`() function and 3 in the `print`() function.

It also uses a shortcut to avoid using the string variable, `m`. It does this by converting the string from `input` to a float in one statement:

```
metres = float (input('\nEnter number of metres: '))
```

Example L3.3: Converting metres to centimetres, version 4

```
# convert4.py: converts metres to centimetres version 3
# Outputs extra blank lines to make it easier to read the output

#Author: Joe Carthy
#Date: 21/10/2023

metres = float (input('\nEnter number of metres: '))

cms = metres * 100

print('\n', metres, 'metres is ', cms, ' centimetres\n\n'  )
```

When you run it, notice the extra blank lines

```
Enter number of metres: 3.5

3.5 metres is 350.0 centimetres
```

Example L3.4: Consider a simple calculator program. This program prompts for two numbers, adds them and displays the sum.

```
# calc.py: Calculator program to add 2 numbers
# Author: Joe Carthy
# Date: 01/10/2023

n1 = float(input('\nEnter first number: '))

n2 = float(input('\nEnter second number: '))

sum = n1 + n2

print('\n\nThe sum of', n1, 'and', n2, 'is', sum, '\n\n')
```

calc.py outputs:

```
Enter first number: 2.4

Enter second number: 5.76

The sum of 2.4 and 5.76 is 8.16
```

More on `print` function and displaying variables

It can get quite complicated when we output several strings and variables using `print` as in the statement

```
print('\n\nThe sum of', n1, 'and', n2, 'is', sum, '\n\n')
```

There is a simpler way to display this message with `print` using **f-strings**. We put the character **f** as the first item in `print` and **we enclose any variable we wish to display in {}** brackets. `print` will display the value of each variable in {}:

```
print(f'\n\nThe sum of {n1} and {n2} is {sum} \n\n')
```

produces identical output to the earlier `print` but is easier to use:

```
The sum of 2.4 and 5.76 is 8.16
```

Displaying a fixed number of decimal places

Python will display the result of numeric calculation to many decimal places.

For example,

```
x = 19/3.768

print(f'x = {x}' )
```

will output on my Mac computer:

```
x = 5.042462845010616
```

In most of calculations it is enough to display result with 2 decimal places.

We use an f-string to do this by following the variable name in {} with the **:.***number of decimal points***f**

So if you wish to display a variables value to 2 decimal places you write `{x:.2f}` to display x to 2 decimal points.

You can change the number from 2 to whatever you wish, to have that number of places displayed after the decimal point.

For example

```
x = 19/3.768
print(f'x = {x:.2f}')
```

outputs

```
x = 5.04
```

Time to practice !

- Copy all of the examples from the slides above and get them to run in your Python environment.
- Then complete the exercises from the Handbook and get them to run.
- Finally carry out the assignments from the Handbook and get them to run.