Introduction to Computer Programming with Python

Table of Contents

Introduction to Computer Programming with Python

9.1 Welcome to Python...

This chapter introduces the concept of computer programming through the medium of Python, an Object Orientated Programming (OOP) language used all over the internet by companies such as Windows, Apple, Facebook, Twitter, Google, Instagram, Youtube, and many, many more.

Python's features include-

1. **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.

2. **Easy-to-read:** Python code is more clearly defined and visible to the eyes.

3. **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.

4. **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

5. **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

6. **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

7. **Databases:** Python provides interfaces to all major commercial databases.

8. Other Files: Python can read all common files; Excel, Notepad, HTML, etc.

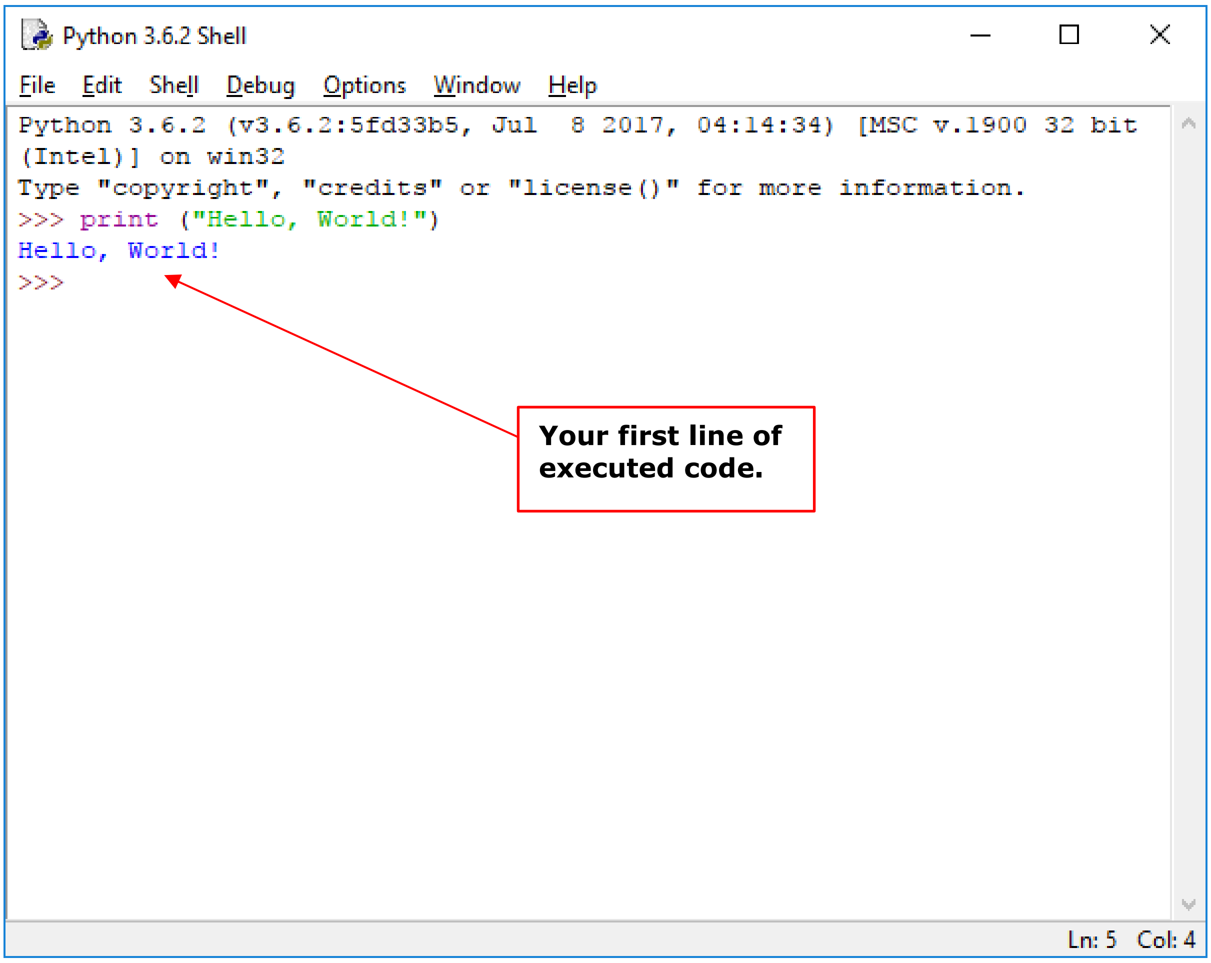
9. **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

9.2 The Python Shell:

The Python command prompt is the three little arrow heads point to the right (**>>>**). This is the command line where we can type Python commands (code). In keeping with a long-standing tradition in computer programming[1], type the following code at the command prompt and then press **[Return]**/**[Enter]**[2];

```
>>> print(“Hello, World!”)
```

The following appears on screen;


Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Your first line of executed code.
Ln: 5 Col: 4

```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:14:34) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print ("Hello, World!")
Hello, World!
>>>
```

Diagram 9.1 – The Python program “Hello, World!” executed in the IDLE shell.

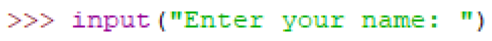
[1] *“Hello, World!” was one of the first programs used to test the C++ compiler back in the early 1907s. To mark this momentous moment, everybody’s first computer program is “Hello, World!”*

[2] *If you receive an error of any type (normally displayed in red), simply start coding again at the command prompt.*

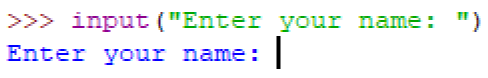
9.4 The `print` Statement:

print(“Hello World!”)

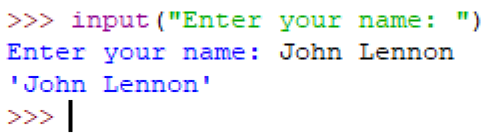
9.5 The `input` Statement:



```
>>> input("Enter your name: ")
```



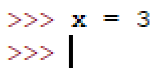
```
>>> input("Enter your name: ")
Enter your name: |
```



```
>>> input("Enter your name: ")
Enter your name: John Lennon
'John Lennon'
>>> |
```

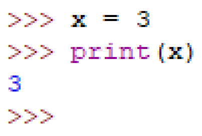
9.6 Assigning Values to Variables:

1.



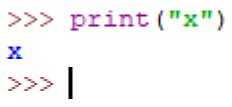
```
>>> x = 3
>>> |
```

Note that when we assign `x` the value of 3, Python does not print the value of `x` on screen. In order to see the value of `x` we must tell Python to print it;



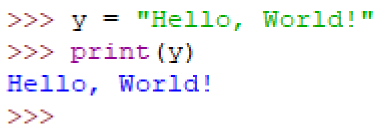
```
>>> x = 3
>>> print(x)
3
>>>
```

2. But;



```
>>> print("x")
x
>>> |
```

3.



```
>>> y = "Hello, World!"
>>> print(y)
Hello, World!
>>>
```

9.7 Variable Names:

`weeksPay = hoursWorked * rateOfPay is better than z = x + y`

Note: camelCase is best and used widely in Python

9.8 Variable Naming Conventions:

Variable names;

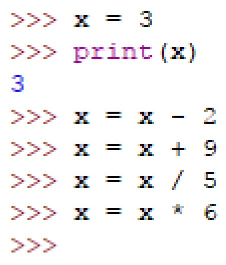
- thisIsAGoodExampleOfCamelcase
- ThisISNot
- Neitheristhis
- thisIs
- itsQuiteSimpleReally
- looksGoodToo
- helpsUsToReadTheCode
- newBankBalance = (CurrentBankBalance + recentDeposits) - recentWithdrawals

Remember: camelCase looks *Pythonic (like Python code should look).*

Note: Python is **case-sensitive**.

9.9 Back to x Again...

We can carry out all the usual mathematical functions on `x`;



```
>>> x = 3
>>> print(x)
3
>>> x = x - 2
>>> x = x + 9
>>> x = x / 5
>>> x = x * 6
>>>
```

Exercise 9.9 – Some Simple Maths Equations:

For each of the following, find `x`;

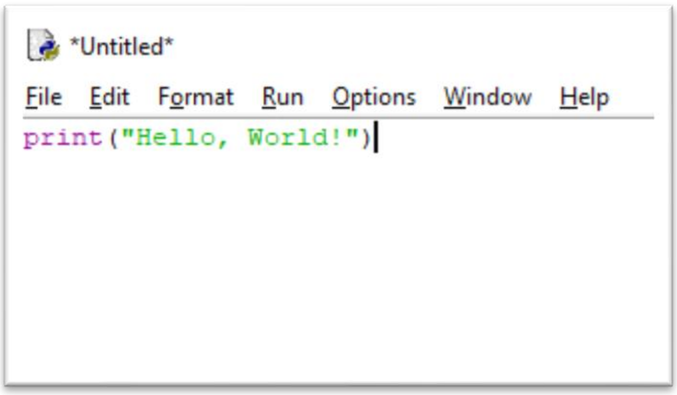
- x = 2 + 3 + 4 + 5
- x = 2 * (3 + 4) – 5
- x = (2 * 3) + (4 * 5)
- x = 2 + 3 + (4 * 5)
- x = 2 * 3 + 4 – 5
- x = 2 + 3 * 4 * 5

As you can see, sometimes maths can be confusing. The simple rule is – *keep it simple*.

9.10 The Python Window (Idle):

Click **File** - **New File**.

Important: Before you continue, create a folder called **Python** in your **My Documents** folder.



```python
print("Hello, World!")
```

After saving your program, you will need to run it; **Run -> Run Module ([F5])**

9.11 Data Types:

Variables can store different types of data;

1. **String:** Anything enclosed in quotation marks; names, text, etc.
2. **Integer:** Whole numbers (e.g. 1, 42, 309)
3. **Float:** Decimals (23.88, 5.99)
4. **Bool:** Logical value (Yes/No, True/False).

```python
x = input("Enter first number: ")

print(x + 3)
```

Or,

```python
x = int(input("Enter first number: "))
print(x + 3)
```

9.12 – Exercise; Variable Assignments:

1. Write a program to accept **two** numbers from the user and add them together before printing the result on screen.

2. Write a program to accept **three** numbers from the user. **Multiply** the first and second numbers and **add** the result to the third number;

   (num1 + num2) * num3

3. Write a program to accept the user's name and then print "Hello John, how are you today". **Note:**
    a. We can add strings together in the same way as numbers;

    ```
    print("Hello " + userName + " how are you today?")
    ```

    b. Or, we can **construct** (make) a new string variable and print that;

    ```
    displayLine = "Hello " + userName + " how are you today?"
    print(displayLine)
    ```

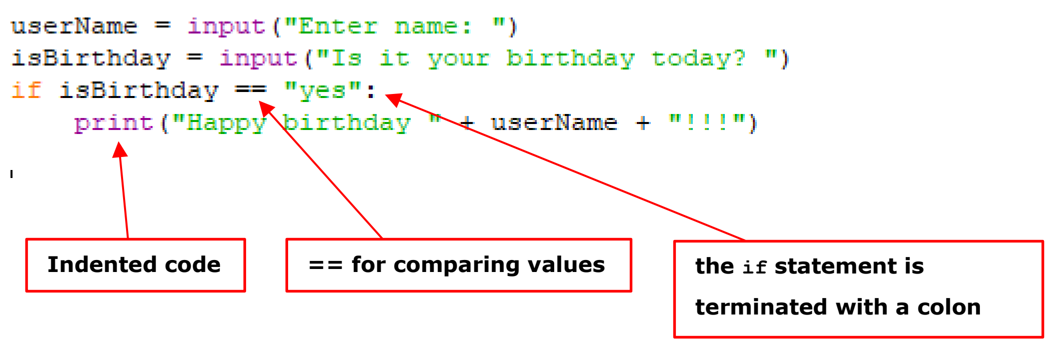
4. Add to the above program by getting the user's age and then print;

    John is 13. (where "John" and 13 are user entries.)

    a. **Note:** we can treat numbers as strings with the `str()` function;

        i. `print(userName + " is " + str(userAge)).`

9.13 The "IF" Statement:



```
userName = input("Enter name: ")
isBirthday = input("Is it your birthday today? ")
if isBirthday == "yes":
    print("Happy birthday " + userName + "!!!")
```

Important:

1. The `if` statement must be terminated by a colon "**:**"
2. Python *indents* (moves in) code that is conditional on the `if` statement.
3. Python uses the double equals sign "`==`" for comparing values.
4. If the user enters "Yes", "YEs", "YES", or anything else that does not match "yes", the `if` condition is considered *false* (not *true*) and the birthday wish is not displayed. This is because we are comparing the value of "yes" to the user input. If the user enters "yes" then Python considers the values to be the same, and the `if` statement true. All code related to the if statement is only executed (run) if the conditions of the `if` statements are true.

9.14 IF Condition Checks;

- == equal to
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

Exercise 9.14 – Compare Two Ages:

Write a program to accept two user names and ages and compare the two ages;

1. Get `userName1`
2. Get `userAge1`
3. Get `userName2`
4. Get `userAge2`
5. If `userAge1` is greater than `userAge2` then print;
   a. John is older than Mary.
6. If `userAge2` is less than `userAge2` the print;
   a. Mary is older than John.
7. Remember that the age variables must be integers.

TIP: You can use `print()` or `print("")` to print a blank line to separate user entries.

9.15 The ELSE Statement:

Your program from the above exercise should look something like;

```
userName1 = input("Enter name 1: ")
userAge1  = int(input("Enter age 1: "))
userName2 = input("Enter name 2: ")
userAge2  = int(input("Enter age 2: "))

if userAge1 > userAge2:
    print(userName1 + " is older than " + userName2 + ".")
if userAge1 < userAge2:
    print(userName2 + " is older than " + userName1 + ".")
```

In programming terms, the above code snippet has two if statements. However, both conditions cannot be true - John cannot be older than Mary at the same time that Mary is older than John. This is where the "ELSE" statement comes in;

```
if userAge1 > userAge2:
    print(userName1 + " is older than " + userName2 + ".")
else:
    print(userName2 + " is older than " + userName1 + ".")
```

Now, if the first condition is true, the second condition is not run. Likewise, if the first statement is false, then the `else` statement is executed.

Note that the code conditional on the `if` and `else` statements are indented. Python is very strict about conditional code being indented. It will give an error if code is not indented properly. There is a very good reason for this. Consider the following pseudocode (computer code written like English);

```
if age1 > age2 then:
      print user1 is older
      print user2 is younger
else:
      print user2 is older
print user1 is younger
```

What would happen if we wrote this program as it is laid out above? What would be the output?

Test 1: *John, 15 and Mary, 12;*

Output from Test 1: *John is older*
Mary is younger
John is younger

Test 2: *John,* ***10*** *and Mary, 12;*

Output from Test 2: *Mary is older*
John is younger

Why? Why did the line "John is younger" print no matter what ages were entered?

Answer: Quite simple really. The last line of the pseudocode above is not indented and therefore is not conditional on the *else* statement. This means that this line is not dependent on the `if` statement. That is why indenting your code properly is so important in Python.

The pseudocode should read;

```
If age1 > age2 then:
      Print user1 is older
      Print user2 is younger
Else:
      Print user2 is older
      Print user1 is younger
```

What about this example?

```
Teacher asks, “Did you do your homework”?
If answer = NO:
      Write note home
Give student detention
```

That poor student is getting detention no matter what he/she says. The correct version should read;

```
Teacher asks, “Did you do your homework”?
If answer = NO:
      Write note home
      Give student detention
```

So, indenting your conditional code is very important. If you get it wrong your program may not do what you expected...

9.16 Back to John and Mary...

What happens in the above code if John and Mary are both 13?

TRY IT YOURSELF...

The program will evaluate the `if` condition and see that it is false (John is *not* older than Mary if they are both 13). It will then jump to the `else` condition and print “Mary is older than John” – a statement we know to be incorrect.

Python lets us check for this *third condition* with the “ELIF”[3] statement;

```
if userAge1 == userAge2:
    print(userName1 + " and " + userName2 + " are the same age.")
elif userAge1 > userAge2:
    print(userName1 + " is older than " + userName2 + ".")
else:
    print(userName2 + " is older than " + userName1 + ".")
```

Our program now catches all possibilities.

Exercise 9.16 – Compare Two Ages Revisited:

1. Add the `elif` condition to your program to compare to two ages.

[3] *“ELIF” is short for “else if”.*

NOTE: Now that your programs are getting a bit more detailed, it is time to start naming them properly so that you can find them when necessary;

a. Name the current program ifElifElse.py. (*Note the camelCase here too?*)
b. Save it to your Python folder.
c. Now you will be able to open the program ifElifElse.py whenever you need to see an example of how to use the `if...elif...else` statement.

9.17 Nested IFs:

The term "*nested ifs*" refers to `if` statements that are placed *inside* other `if` statements. An example of this would be;

```
schoolDay = "YES"
eightOClock = "YES"

if schoolDay == "YES":
    if eightOClock == "YES":
        print("Get up for school.")
else:
    print("Stay in bed.")
```

The second `if` statement above is said to be *nested inside* the first. Therefore, the code checking for eight o'clock only occurs if it is a school day. If it is not a school day, then there is no point in checking the getup time.

Nested `if` statements can be very useful. They can be used to control the flow of logic through a program. But, consider the following;

```
if schoolDay == "YES" and eightOClock == "YES":
        print("Get up for school.")
else:
    print("Stay in bed.")
```

Technically this code is correct (and is actually quicker to process than the previous version.) But when would we ever get to stay in bed? When would the `else` print? What is the opposite of school day *and* eight o'clock?

Whatever the answer, it is not straightforward. So, while nested `if` statements can be a very clever way of programming, we must make sure they do what we intended.

Exercise 9.17 – The Doctor Will See You Now...:

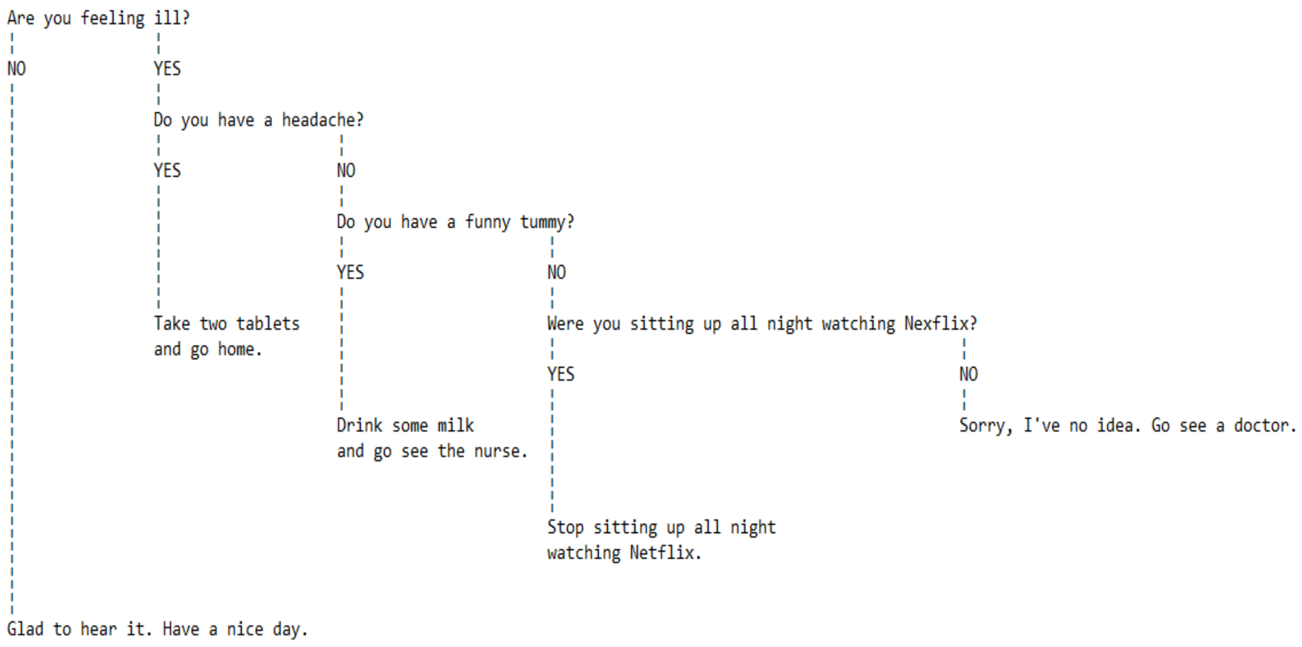
This is a great exercise to test your nested ifs and indenting skills. The idea is quite simple – ask a patient a number of questions until you find out what is wrong with them. Here's a sample output;

```
Are you feeling ill [YES/NO]? YES
I'm sorry to hear that. Do you have a headache [YES/NO]? NO
Okay..., Do you have a funny tummy [YES/NO]? NO
Were you up all night watching Netflix? YES
Stop sitting up all night.
```

The questions and remedies are up to you to decide upon...

STEPS;

1. Ask the user are they poorly.
2. If the user replies "YES, ask another question to try to find out what is ailing them. When the problem is found, suggest a remedy.
3. This program is based on multiple nested `if` statements. We could graph this as follows;


Are you feeling ill?
NO
YES
Do you have a headache?
YES
NO
Do you have a funny tummy?
YES
NO
Take two tablets and go home.
Were you sitting up all night watching Nexflix?
YES
NO
Drink some milk and go see the nurse.
Sorry, I've no idea. Go see a doctor.
Stop sitting up all night watching Netflix.
Glad to hear it. Have a nice day.

4. The more questions you ask, the harder the program will be to code, so start simple and continue adding questions.
5. Do not spend too long on this exercise. Its general purpose is to highlight the functionality of nested `if` statements, not to waste your time.

9.18 A Quick Recap:

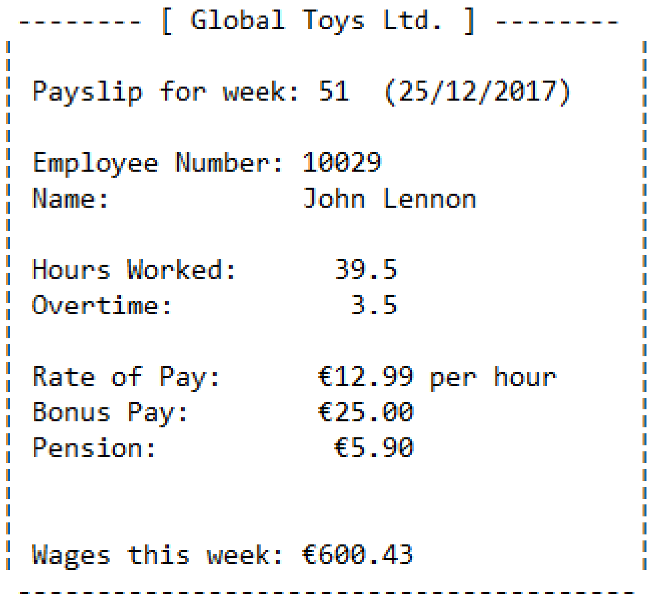
At this stage of developing your programming skills, you can;

1. Print stuff on screen.
2. Get information from a user.
3. Get information from a user and print it on screen.
4. Write conditional statements (`if:.../elif:.../else:...`).
5. Write nested conditional statements (*nested ifs*)
6. Compare numerical data (`if x == y:`).
7. Carry out basic mathematical functions on numeric data (`x = y * 1.5`).
8. Convert strings to integers with the `int()` function, and integers to strings with the `str()` function.
9. Build strings (e.g. *x = "Hello " + userName + ", how are you today?"*)

So, this would be a good time to recap on our new-found coding skills with the following exercise;

Exercise 9.18 – Payroll:

Write the program *payroll.py* to create the following payslip;



```
-------- [ Global Toys Ltd. ] --------
|                                      |
| Payslip for week: 51  (25/12/2017)   |
|                                      |
| Employee Number: 10029               |
| Name:            John Lennon         |
|                                      |
| Hours Worked:      39.5              |
| Overtime:           3.5              |
|                                      |
| Rate of Pay:      €12.99 per hour    |
| Bonus Pay:        €25.00             |
| Pension:           €5.90             |
|                                      |
|                                      |
| Wages this week: €600.43             |
---------------------------------------
```

1. Get the required values from the user;
    a. Employee name
    b. Hours worked

   c. Overtime hours worked
   d. Overtime rate
   e. Rate of pay
   f. Bonus payment
   g. Pension deductions
   h. Do not ask the user for the employee number, week number, or today's date. Just make these up and print them on the payslip.

2. Naming Variables:
   a. Ensure that you name your variables so that they represent the data being stored. For example; `empName`, `hoursWorked`, etc.

3. Calculations:
   a. Overtime rate = rate of pay * 1.5 (*this is called "time and a half"*).
   b. Bonus pay is to be added to wages.
   c. Pension deduction is to be subtracted from wages.

4. Printing payslip;
   a. Each line of the payslip can be printed with the print statement;

```
print("¦ Name:        " + username + "        ¦")
```

   b. Or assigned to a print variable that is then printed;

```
printLine = "¦   Name:        " + username + "      ¦"
print(printLine)
```

   However, do not spend too much time on the payslip boarders.

5. The program should be written in 3 sections;
   a. Section 1 = get user input
   b. Section 2 = do calculations
   c. Section 3 = print payslip

6. Example of how to print payslip;

```
print("  -------- [ Global Toys Ltd. ] --------")
print("¦                                       ¦")
print("¦ Payslip for week: 51 (25/12/2017)     ¦")
print("¦                                       ¦")
print("¦ Employee Number: 10029                ¦")
print("¦ Employee Name: " + empName + "        ¦")
```

etc.....

9.19 Loops in Python:

The concept (idea) of loops is extremely important in computer programming. The basic idea is that we tell the computer to keep doing the same thing *until* something happens, and then stop doing it. We see this all the time in computer games where we are asked if we would like to play again. If we say yes, the game starts again. If we say no, the game finishes.

There are two main types of loops in computer programming;

1. Do stuff until a condition is met that stops the loop – the `while` loop
2. Do stuff for a set number of times - the `for` loop

9.20 Before We Continue - The BOOL Data Type:

As mentioned earlier, Python has a `bool` data type. The term bool is short for *Boolean*. A Boolean is a variable that can only ever have one of two possible values; True or False. Because a `bool` can only ever be True or False, we can use `bool` variables to check for yes/no style answers; "Start Again?", "Do you wish to leave?", etc;

```
x = input("Do you wish to continue? ")

userAnswer = False

if x == "YES":
   userAnswer = True

if userAnswer is True:
     print("User said continue.")
else:
     print("User said end.")
```

In the above code snippet, we asked the user if they wished to continue. If yes, we set `userAnswer` to be *True* – if no, we set `userAnswer` to *False*. The program then carries out some action code based on the value of `userAnswer`.

Note:

1. Note that "True" is spelled with a capital "T" in the `if` statement. This is the correct syntax for checking the value of a `bool` variable. The other possible value would be "False", spelled with a capital "F".
2. The `if` statement uses the keyword `is` to check the value of `userAnswer`. This is a shorthand version of saying `if userAnswer == True:` Another way to check the value of `userAnswer` is as follows;

```
if userAnswer:
     print("User said continue.")
else:
     print("User said end.")
```

Both shorthand versions are correct, but the `is True` version is easier to follow in very complex coding.

TRY IT YOURSELF…

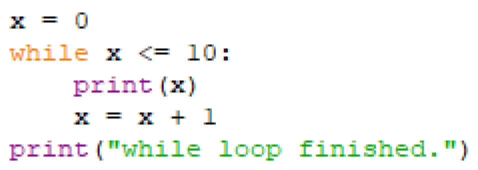
REMEMBER: Python defaults user entries to the *string* data type. That is why we assign the bool variable a value based on the user input.

9.21 The `WHILE` Loop:

The `while` loop tells Python to continue doing something until we tell it to stop. `while` loops normally involve some sort of user interaction to stop them ("play again [Y/N]?"). The syntax (correct way of writing) of the `while` loop is as follow;

while <condition>:
    code to keep doing until the while condition no longer holds true

Python evaluates (checks) the `while` loop condition every time it loops around. The loop ends if the condition of the `while` loop is false. Try the following;



```
x = 0
while x <= 10:
    print(x)
    x = x + 1
print("while loop finished.")
```

TRY IT YOURSELF…

This code sets `x` to 0 and then loops around the `print(x)` statement *while* `x` is less than or equal to 10. When `x` becomes 11, the condition of the while loop is no longer *true*, so the loop ends.

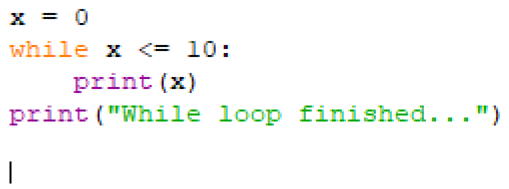
Note: It is very important that `x` is incremented (increased) by 1, otherwise the `while` condition would be true forever and the program would run forever. (If this happens, press **[Ctrl-C]** to stop the Python shell).

Exercise 9.21 – The `while` Loop:

Write different versions of the above while loop and note the output (stuff printed on screen). Notice how the printed value of `x` can change depending on where you place the line `x = x + 1`. Try different number ranges for `x`, but be careful that you do not get stuck inside an infinite loop (a loop that goes on forever).

9.22 The Infinite Loop:

Consider the following;

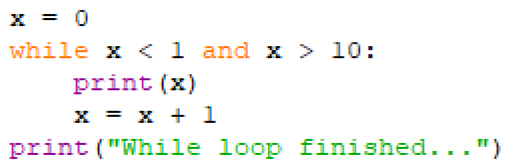


```
x = 0
while x <= 10:
    print(x)
print("While loop finished...")
```

What happens when this code is run?

Initially, control enters the loop because the condition of the loop is true (`x` is less than or equal to 10). Once in the loop, control loops around and around, never getting out because x is always 0. This is known as an *infinite loop*.

What about the following;



```
x = 0
while x < 1 and x > 10:
    print(x)
    x = x + 1
print("While loop finished...")
```

What happens when you run this code? **TRY IT YOURSELF...**

The output from this code reads; "While loop finished". This is because the loop is never entered in the first place. To see why, we need to break down the `while` loop statement. Firstly, the `while` loop statement has 2 conditions;

    while x < 1 ***and*** while x > 10

We can clearly see that the first condition is true because `x` is set to 0. However, the other condition for entering the loop is that `x` > 10. But `x` is 0 and therefore is not greater than 10. Therefore, the while loop is never entered.

Simple mistakes like these can catch us time after time. Whether it is an infinite loop or a loop that will not run. Just remember to check the conditions of your loops.

9.23 Breaking Out of Loops:

Python provides the keyword `break` to break out of loops. Consider the following;

```
loopCounter = 0
userAnswer = "YES"
while userAnswer == "YES":
    loopCounter = loopCounter + 1
    print("Testing a while loop: " + str(loopCounter))
    userAnswer = input("Loop again? ")
    if userAnswer == "NO":
        break

print("While loop finished...")
```

TRY IT YOURSELF...

Note: Do not worry about the user entering the invalid replies.

The above loop terminates (stops) when the user replies "NO" to the question "Loop again?". When the user replies "NO", the following `if` condition is *true* and the `break` statement occurs. Otherwise, the loop continues.

This is the type of program control that encapsulates many computer applications, apps, games, etc. The loop continues until the user says/clicks No.

Exercise 9.23 – Payroll Revisited:

1. Add a controlling loop around *payroll.py* that asks the user if they wish to print another payslip. If yes, repeat the code. If no, end. Remember to save your program.
2. Python expects dependent code to be indented. This means that you will have to indent all the code in the controlling `while` loop. We can use the keyboard shortcut of **Ctrl-]** to quickly indent a block of code;
   a) Highlight the code you want to indent
   b) Press and hold down the control key (**[Crtl]**)
   c) With Ctrl pressed down, press the "close square bracket key"; **]**
   d) This will indent a block of code.
   e) Use the keyboard shortcut **Ctrl** + **[** to unindent a block of code.

9.24 The Long String Data Type:

As well as the data types we have already seen, Python also has a data type called the *long string*. The long string data type is used for printing very long pieces of text or to assign extremely long pieces of data to a variable. Try the following – or something similar;

```
print('''
This is a very long piece of text.
It goes on for ages.
Line after line.
It seems to never stop.
Is there no limit to the amount of data a Python string can hold.
I could literally be here all night...
''')
```

TRY IT YOURSELF…

Notes;

1. The syntax for the long string in the triple single quotation mark (''')

2. The triple single quotation mark must appear at the start and end of the long string.

3. Notice how Python keeps the screen format of what you type in. Python stores (but does not display) the [Return]/[Enter] keystrokes in the long string and then applies them when it displays the long string.

4. The long string can also be used for variable assignment;

```
screenDisplay = '''
This is a very long piece of text.
It goes on for ages.
Line after line.
It seems to never stop.
Is there no limit to the amount of data a Python string can hold.
I could literally be here all night...
'''
print(screenDisplay)
```

5. While this is a great way to write documents and HTML pages on the fly (from a program that is running), it can also be used to create an opening screen for a program.

Exercise 9.24 – Payroll Revisited:

Create an opening screen for your payroll system. For example;

```
***  **  *   *  ***   ***   *    *
* * *  *  * *   * *  *   *  *    *
*** ****   *    ***  *   *  *    *
*   *  *   *    **   *   *  *    *
*   *  *   *    * *   ***   **** ****
WELCOME TO PAYROLLS ARE US!!!

We take care of all your payroll needs while you take care of your business. Our payroll
system allows you to... (continue from here)

INSTRUCTIONS:
Follow the instructions on screen to enter the details of each employee. When the details are
successfully entered into PAYROLLS ARE US, a payslip will be printed with the correct deductions
and addictions... (continue from here)
```

Note: *Ensure that the opening screen is outside the while loop.*

9.25 A Quick Comment:

One of the most important parts of computer programming is adding comments to your code so that both you and fellow programmers know what is going on.

Python has to types of comments;

1. Single line comment
2. Multiline comment

9.25.1 Single Line Comment:

Python uses the hash symbol, **#** to mark a single line comment;

```
# This is a single line comment
```

Single line comments are used to explain what the following code does;

```
# Calculate employee's wages
employeeWages = (hoursWorked * rateOfPay) + overTime + bonusPayment - pensionDeductions
```

9.25.2 Multiline Comments:

Multiline comments are comments that continue over more than one line and are normally used to explain what a particular section of code does. Python uses the same syntax for the multiline comment as the long string data type – the triple single quotation mark;

```
'''
The following code asks the user for the employee
details and then prints a payslip for that employee.
The user can repeat the process if needs be.
For now the payslip is being displayed on screen, but
soon it will be sent to a text file.
'''
```

Note that the multiline comment needs to be terminated (ended) with the same triple single quotation mark.

Exercise 9.25 – Payroll Revisited:

Open *payroll.py* and add useful comments so that any other Python programmer will know exactly what is going on at any time. Remember, this is a very important part of programming in any computer language. The lack of appropriate commenting is seen as bad programming practice.

9.26 – The FOR Loop:

As said earlier, the `for` loop is used to do something an exact number of times. If we know we what to loop a piece of code 5 times, we would use a `for` loop instead of a `while` loop.

TRY IT YOURSELF...

Try the following and see what prints. (Be careful, note the indentation!!)

```
for x in range(1, 10):
    print(x)
print("for loop finished.")
```

Firstly, note that `x` does not have to be declared before it appears in the `for` loop.

Secondly, most of us would have expected that the numbers 1 to 10 would have printed on screen. Instead, the last number that was printed was the number 9. This is because the `for` loop evaluates the loop condition at the start of each iteration. So, the first time in `x` is 1, next time round `x` is 2, 3, 4 and so on until x becomes 10. When `x` becomes 10 the for loop stops, thus never reaching the print statement to print 10.

While this could be seen as a problem, at least we know it will happen every time we use a `for` loop and as such as can adjust our code accordingly;

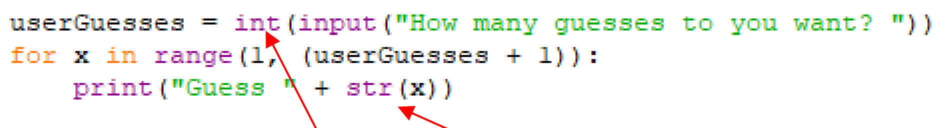
```
for x in range(1, (10 + 1)):
    print(x)
```

Syntax of For Loop:

1. Like the `while` statement, the `for` statement *must* be terminated with a colon.
2. The `range()` function uses a comma separated list of numbers; `range(1, 5)`.
3. Note that the `for` statement uses the keyword *in*.

4. Note that the `for` statement declares the variable `x` for us, we do not need to declare it before we use it.
5. Dependent code *must* be indented.

A better example would be to ask the user how many times they want to guess an answer and then use a `for` loop to iterate that many times;



```
userGuesses = int(input("How many guesses to you want? "))
for x in range(1, (userGuesses + 1)):
    print("Guess " + str(x))
```

Remember;

1. We must use the `int()` function to convert the user entry to an integer.
2. We must convert the integer to a string with the `str()` function if we want to add it to a string.

TRY IT YOURSELF…

Exercise 9.26 – The `for` Loop:

1. Write a program that asks the user for a number and then print that number that amount of times. For example, if the user enters 9, print 9 nine times.

2. Write a program that asks the user for a number and then print a descending list of numbers starting with the user's entry. For example, if the user enters 9, print 9, 8, 7, etc. down the screen.

   Hint:

   i. `x` increases by 1 every time around the loop
   ii. we know the user's number
   iii. we can create a mathematical calculation that decreases by one from the user's number and the print it; `y = userGuesses - x`
   iv. This is a common programming task, to manipulate numbers based on values that we have.

3. Change the above program to print a list of the squares of the descending numbers. For example, if the user enters 5, print 25, 16, 9, 4, 1 down the screen. (Note: To "square" a number means to multiply it by itself [2 squared is 2 * 2, 4)]

9.27 The `LIST` Data Type:

As its name suggests, the `list` data type is a comma separated list of values;

```
daysOfWeek = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
print(daysOfWeek)
```

Syntax of the List Data Type;

1. A list must be enclosed in square brackets; **[…]**
2. Each element in a list must be separated by a comma; “Monday”, “Tuesday”
3. Elements in a list can be of any data type; [“Monday”, 33, 89.99]
4. Sample list; seasons = [“Spring”, “Summer”, “Autumn”, “Winter”]

Lists are a very powerful tool in Python. For example,

1. We can print an element from a list based on its position; `print(seasons[2])`
2. We can change an element in a list; `seasons[2] = “Fall”`
3. We can get the number of elements in a list with the `len()` function; `x = len(seasons)`
4. We can *slice* lists; `print(listWeekdays[2:5])`
5. We can add an element to a list with the `append()` function;

   `seasons.append(“New Season”)`
6. We can insert an element into a specified index point;

   `Weekdays.insert(2, “Newday”)`
7. We can add a list to a list with the `extend()` function;

   `myFavouriteThings.extend(“School”,”Homework”,”Detention”)`
8. We can delete elements from a list with the `del` function;

   `del seasons[2]`

IMPORTANT: The position of an element in a list is known as its *index*. Index values are integers and start at 0. Therefore, the first element in a list has an index value of 0, not 1 as you might expect.

Code:

```
seasons = ["Spring", "Summer", "Autumn", "Winter"]
print(seasons)
print("Season at index 0 is ", seasons[0])
```

Output:

```
['Spring', 'Summer', 'Autumn', 'Winter']
Season at index 0 is  Spring
```

So, be careful with lists, especially lists of numbers;

```
My list of numbers:  [1, 2, 3, 4, 5]
The number at index 0 is  1
...
```

As we can see from the above output, the number 1 had an index value of 0. This can be confusing and can cause errors in programs. A common runtime error (an error that occurs when you run your program) associated with lists is the *list index out of range* error;

```
    print("The number at index 0 is ", listOfNumbers[6])
IndexError: list index out of range
```

This error occurs when we try to reference an index position beyond the length of the list.

Exercise 9.27 – Some `LIST` Programs:

1. Write the following program;
   a. Create a Python list of the days of the week.
   b. Ask the user to enter a number.
   c. Print the corresponding week day in a sentence as follows;
      i. Day 3 is Wednesday.
      ii. **Hint:** You can use the `userNumber` as the index value;
          `weekDays[userNumber]`

2. Write a `for` loop program to print the days of the week down the screen.
   a. **Hint:** `for x in range(0, len(weekDays))`
   b. **Remember:** List index values start at 0.

3. Write the following program;
   a. Create a Python list of the months of the year.
   b. Ask the user to enter a number.
   c. Print the corresponding month in a sentence as follows;
      Month 10 is October.
   d. **Remember:** List index values start at 0, so month 10 in the list will be November. Make sure your program corrects this.

4. Write the following program;
   a. Create an *empty* Python list (`listName = []`).
   b. Ask the user to enter their name.
   c. Ask the user to enter 5 things they like one at a time.
   d. When the user has entered 5 things, print them down the screen.
   e. Here's the screen output;

```
Please enter your name: Matt
Enter a thing your love: bats
Enter a thing your love: cats
Enter a thing your love: hats
Enter a thing your love: mats
Enter a thing your love: rats
Matt likes....
bats
cats
hats
mats
rats
```

9.28 Keep Printing on the Same Line:

So far all of our screen output has printed down the screen. This is because Python adds a new line (**\n**) character to the end of every `print()` statement. However, we can tell Python not to add the new line character to the end of the `print()` statement and to continue printing on the same line;

```
print("Hello ", end = "")
```

The `end = ""` tells Python not to end the `print()` with "" instead of (\n) at the end of the print statement. By replacing the new line with a null (empty) value, Python continues to print on the same line. So, the following code;

```
userName = "John"

print("Hello ", end = "")
print(userName, end = "")
print(", how are you today?")
```

Prints;

```
Hello John, how are you today?
>>>
```

This can be very helpful when printing from inside a loop;

```
for x in range(0,5):
    print(favouriteThings[x], end = ", ")
```

Here we have replaced the new line character (`\n`) with a comma and a space; ", ". This tells Python to print a comma and a space at the end of the `print()` statement. Next time around the `for` loop, Python continues printing on the same line.

Exercise 9.28 – Keep Printing on Same Fine:

Return to your previous program and make Python print the user's 5 favourite things on the same line. (Do not worry about the last comma, we will clean that up soon.)

9.29 The `TUPLE` Data Type:

A `tuple` in Python (and in many other OOP languages) is very similar to a `list`, except that we cannot change the values of a tuple. We cannot add elements to, remove elements from, or overwrite elements in a `tuple`. And we cannot append values to a `tuple`.

The `tuple` data type is declared as follows;

```
        weekDays = ("Mon", "Tues", "Wed", "Thurs", "Fri", "Sat", "Sun")
```

Note that a `tuple` is declared by using *curly* brackets instead of the *square* brackets of a `list`.

Tuples are normally used for lists of fixed length; days of the week, months of the years, etc. We can find elements in tuples as this does not change the tuple. Similarly, we can use the `len()` function on tuples. But, we cannot use the `.append()` or `.expend()` functions.

The general rule is; if you do not intend on changing the values in your list of elements, use a `tuple`. Otherwise, use a `list`.

Technically, because tuples are of fixed length they are much faster to access. While speed is not a major concern for us now, good programming standards are very important.

Exercise 9.29 – Print Month of Year with Tuple:

Write a short program that asks the user to enter a number and then, using a tuple, print the corresponding month as follows;

```
        Month 10 is October.
```

9.30 Functions:

So far all of our program code has executed from the top of our programs to the bottom. This is known as *procedural programming*. Procedural programming is easy to follow as it runs down the page. But, as we have seen many times, there is a lot repetition of code in procedural programming.

Functions allow us to write code once that we can call whenever we need it. For example, if we needed the user to input an integer (e.g. `userAge`), we could write a function for getting any integers value from the user, and then just call it. Similarly, we could write a Yes/No function that we can call every time we want the user to answer a question.

Functions are just pieces of code. They are written at the top of a program and are declared with the keyword "`def`". Functions must have a unique name to identify them, followed by a set of curly brackets and a colon; e.g. `def getInteger():` Just like the `if`, `while` and `for` statements, dependent code must be indented. Functions can do anything we want them to do and we can call them as often as we wish.

Techo note: Functions only exist when they are called. If we write a 1000-line function in a program and never call it, it does not exist in terms of using up computer memory. As such, functions not only save us from writing lines and lines of code over and over again, they also speed up our programs. Here's a quick example;

1. Write a normal program that prints the popular happy birthday greeting for John.
2. Here's the output;

```
Happy birthday to you.
Happy birthday to you.
Happy birthday dear John.
Happy birthday to you.
```

9.30.1 Our First Function:

As we can see, our program consisted of the same line being printed 3 times. This is perfect for a function;

```
def printLine():
    print("Happy birthday to you.")

printLine()
printLine()
print("Happy birthday to John.")
printLine()
```

Okay, not a great use of a function, but it demonstrates our point pretty well;

1. The line `def printLine():` declares our function with the name `printLine`.
2. All the code relating to the function is indented.
3. We call the function by its name.
4. When the function is called, Python executes the code in the function *as if we had written the code at the point of calling it*.
5. The output from this program is identical to the previous happy birthday program.

What happens if it is Mary's birthday too? Or maybe Paul's and Paula's too?

We can modify our function so that we can call it and send it the person's name;

```
def happyBirthday(name):
    print("Happy birthday to you.")
    print("Happy birthday to you.")
    print("Happy birthday to " + name + ".")
    print("Happy birthday to you.")

happyBirthday("John")
```

Now the function `happyBirthday()` is set to accept the parameter (an extra piece of information) "**name**". When the function is called, the parameter "name" is replaced with the extra piece of information we send to the function; "**John**".

Now, if it's Mary's birthday we can simply call the `happyBirthday()` function and send it Mary's name instead.

```
happyBirthday("Mary")
```

Now we can begin to see how functions can save us time writing the same code over and over again, as well as speeding up our programs.

Note If your function expects a parameter and none is sent, it will give a *runtime* error.

TRY IT YOURSELF…

Note: If your function expects no parameters and one (or more) is sent, it will give a runtime error.

TRY IT YOURSELF…

Exercise 9.30.1a – Happy Birthday…:
Using the code snippets from above;

1. Write a program that uses the `happyBirthday()` function to print the popular happy birthday greeting on screen.
2. Call the function to wish **John** a happy birthday.
3. Change your program to accept a name from the user and pass that name to the `happyBirthday()` function.
4. Place a `while True:` loop around the program so that the user can keep entering names to wish a happy birthday to. Make sure the function declaration is not inside the loop. Use Ctrl-C to break out of your looping program for now.

Exercise 9.30.1b – Maths Functions:
Write the following programs;

1. Write a program that uses a function to add to numbers and prints the result.
   STEPS:
   a. Declare your `mathsAdd()` function at the top of your program.
   b. Call your function, passing it two numbers; `mathsAdd(2,3)`

   c. The function must add the two numbers and print the result on screen.
   d. Change your program to get the two numbers from the user.

2. Add a function called `mathsSubtract()` that subtracts one number from another.

   STEPS:

   a. Copy the function `mathsAdd()` to `mathsSubtract()`
   b. Change the mathematical operator
   c. Change the screen output if required.
   d. Call your new `mathsSubtract()` function and check the results.

3. Add two more functions to your program;
   a. `mathsMultiply()`
   b. `mathsDivide()`
4. Save your program as `mathsFunctions.py` in your python folder.

9.30.2 Maths Functions Revisited:

Did you notice anything while writing the above program? Did you notice how you kept doing the same thing over and over again? Look back at your code. Does it look repetitious? This idea of doing the same thing over and over again should be the spark for us to write a function.

But, we've just written four functions (+, -, *, /) to save space and time!!!

Agreed, but the four functions did pretty much the same thing except for the mathematical operator. What about if we passed the mathematical operator to a function as well, and then the function could decide what to do?

```
mathsFunction(2, 3, "+")
```

Or,

```
mathsFunction(2, 3, "*")
```

Writing one function to do all four mathematical operations would save us from writing four different functions.

Our `mathsFunction()` would simply except a third parameter of "+", "-", "*", or "/". Our `mathsFunction()` would then decide what to do based on the third parameter;

```
def mathsFuction(num1, num2, vOperator):
    if vOperator == "+":
        z = num1 + num2
        print(num1, num2, z)
    elif vOperator == "-":
        z = num1 - num2
        print(num1, num2, z)
```

Exercise 9.30.2 – Complete `mathsFunction()`:

Using the above code snippet, modify your program *mathsFunctions.py* to use one function to carry out the four basic mathematical operations. Test the logic flow of your program by calling your function with different values and math operators.

Techo Note: Some might argue that calling `mathsFunction()` every time we need to add two numbers could be a waste of computer memory (speed). This is most probably true. However, this exercise is about understanding the power of functions, not speeding up our programs.

9.30.3 Returning a Value from a Function:

So far, all of our functions have accepted a number of parameters to work with, and have given us nothing in return. However, functions can also pass back a value. To make our maths functions more useful, we can get them to pass back the answer rather than print it on screen;

```
#mathsAdd = add two numbers and pass back the answer
def mathsAdd(num1, num2):
    z = num1 + num2
    return z

answer = mathsAdd(2, 3)
print(answer)
```

This code snippet introduces the keyword **`return`**. The keyword `return` tells a function to send the value of `z` back to the calling code. Note that we did not need to pass `z` to the function. Once we use the keyword `return`, the function *knows* it must send the corresponding value back to the calling code.

The above `mathsAdd()` function adds to given numbers and passes the result back to the variable `answer`. Because the result of the function is now back in the main program, we can now do something with it. This idea of passing data to functions and back from functions is a very important part of using functions in Object Orientated Programming.

Exercise 9.30.3 Return a Value from a Function:

With the above in mind, write the following program;

1. Accept two numbers and a mathematical operator from the user.
2. Pass the values to `mathsFunction()`
3. Modify `mathsFunction()` so that it passes back the result of the action.

4. Assign the value passed back from `mathsFunction()` to the variable `answer`.
5. After the function call, print the results to screen as follow;

   2 + 3 is 5

   2 – 3 is -1

   2 * 3 is 6

   2 / 3 is 0.666666666
6. Required values;
   a. `userNum1`
   b. `userNum2`
   c. `userAction`
   d. `answer` (value passed back from function)
7. Print line example;

   ```
   print(str(userNum1) + " " + userAction + etc... )
   ```

9.30.4 Remember our Payroll Program...?

With the knowledge of how functions work, we could rewrite our payroll program from earlier as follows;

```
openDatabase(dbOracle)
while True:
    readEmployee()
    calcWages(employeeNumber)
    printPayslip(employeeNumber)
closeDatabase(dbOracle)
```

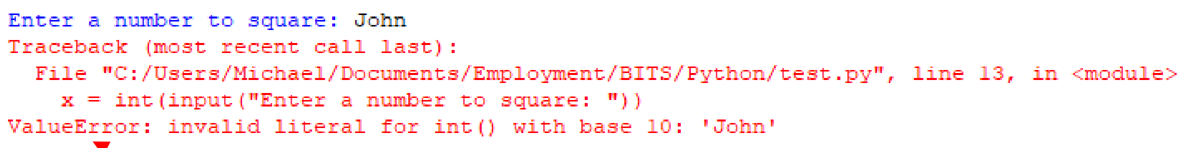
This code snippet gets its data from an Oracle database[4] instead of the screen, reads each employee record, calculates the wages of each employee and then prints a payslip before doing it all over again for the next employee in the database. Such a program could print thousands of payslips in less than a minute.

This is the power of functions in computer programming. They keep our code tidy. They speed up our programs. They save us from writing the same code over and over again. Your first thought when writing code should be, "*Can I use a function to do that?*". But remember, functions must be useful.

[4] *A database is just a collection of organised data. Databases can hold information on customers, employees, suppliers, etc. This data can then be accessed and acted on by programming languages like Python.*

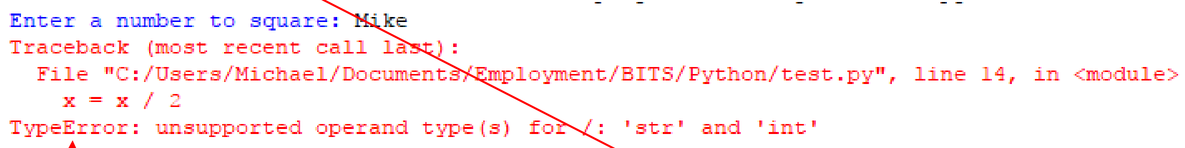
9.31 Data Validation with Functions:

By now you will have seen many Python error messages displayed on screen. An error message occurs when the user enters an integer value instead of a string value, while a different error message occurs when the user enters a string value instead of an integer value;



```
Enter a number to square: John
Traceback (most recent call last):
  File "C:/Users/Michael/Documents/Employment/BITS/Python/test.py", line 13, in <module>
    x = int(input("Enter a number to square: "))
ValueError: invalid literal for int() with base 10: 'John'
```

Or;

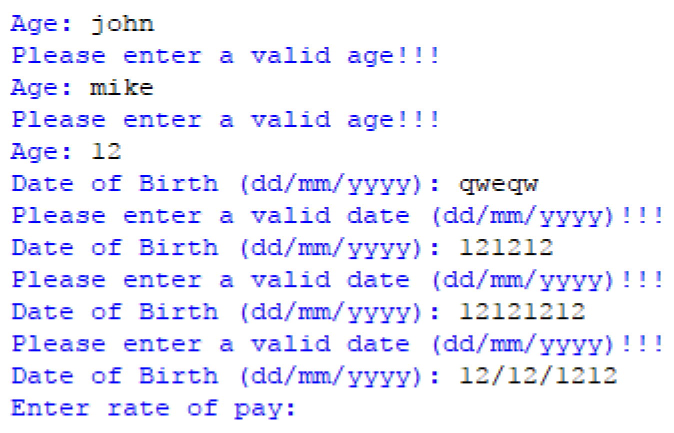


```
Enter a number to square: Mike
Traceback (most recent call last):
  File "C:/Users/Michael/Documents/Employment/BITS/Python/test.py", line 14, in <module>
    x = x / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Data validation is when we force the user to enter the correct data type that we are expecting. Put simply, we keep asking the user to enter a value until they enter the correct type. In computer programming, this is called *error handling*.

We, as computer programmers, can catch all types of errors. We can catch *ValueErrors*, *TypeErrors*, and many more.

The normal procedure for *error-trapping* is to place a `while` loop around the `input()` statement and *force* the user to keep entering a value until they get it right. Here's an example of error handling;



```
Age: john
Please enter a valid age!!!
Age: mike
Please enter a valid age!!!
Age: 12
Date of Birth (dd/mm/yyyy): qweqw
Please enter a valid date (dd/mm/yyyy)!!!
Date of Birth (dd/mm/yyyy): 121212
Please enter a valid date (dd/mm/yyyy)!!!
Date of Birth (dd/mm/yyyy): 12121212
Please enter a valid date (dd/mm/yyyy)!!!
Date of Birth (dd/mm/yyyy): 12/12/1212
Enter rate of pay:
```

As we can see in the above screen output, the user entered *john* and *mike* before entering a valid age of *12*. They then entered *qweqwe*, *121212*, *12121212* before entering a valid date of birth of 12/12/1212.

Each time the user entered an *invalid* entry, the program warned them with an error message, and then asked them to re-enter the required value.

Here's the code;

```
def inputInt(vQuestion, vError):
    while True:
        try:
            vInput = int(input(vQuestion))
        except ValueError:
            print(vError)
            pass
        else:
            return(vInput)

inputInt("Please enter your age: ", "Invalid age entered, please try again!")
```

There is a lot going on in this code snippet, so we will take some time to break it down;

1. Firstly, we declared our function `inputInt()`

2. Our function `inputInt()` takes two parameters, `vQuestion` and `vError`. Instead of writing the question "Please enter your age: " *in* the function, the function accepts the question as a parameter. The same is true of the error message. This way we can use the `inputInt()` function anytime we want the user to enter a number – we simply call the function and change the parameters to suit the input we want;

   ```
   inputInt("Enter a number to square:", "Please enter a number!!!")
   ```

3. The function `inputInt()` uses a `while True:` loop to hold the user in the function until they enter a numeric value.

4. The function also uses two new Python concepts;
   a. `try...except...else`
   b. `pass`

Let's take a look at these in detail.

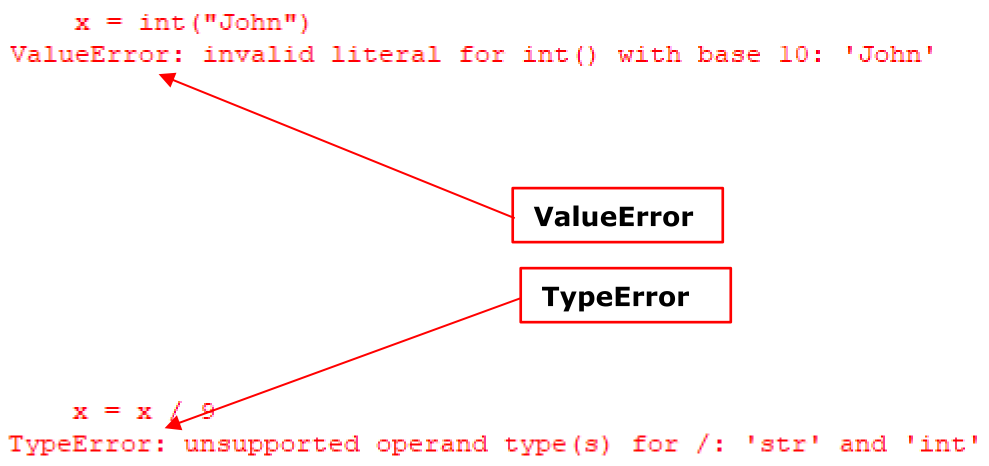
9.31.1 Try...Except...Else:

The keywords `try...except...else` can be best explained in English as follows; you can try to enter any value you want, but you're staying here until you enter the data type *I* want.

The `try` section contains the input statement. This asks the user to enter their age.

The `except` section catches the error; `Please enter a valid age!!!` The keyword `except` is an abbreviation of *exception*, and is executed when Python tries to display an error (an exception) on screen. By using `except` we can catch Python runtime error messages *before* Python displays them on screen.

There are many different types of errors in Python (ValueError, TypeError, etc.). Python displays the type of error on screen when an error occurs;



```
    x = int("John")
ValueError: invalid literal for int() with base 10: 'John'
```

```
    x = x / 9
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

The above *ValueError* occurs when we try to convert *John* to an integer. The *TypeError* occurs when we try to divide *John* by 9.

We can use the error information with the keyword `except` to catch the error;

```
except ValueError:
    print(vError)
    pass
```

The keyword `pass` tells Python to go straight back up to the `while True:` loop and start again.

The last section of `try...except...else` is the else section. The `else` section is run when we have the data type (integer) that we want. The `else` section leaves the function and returns the value entered to the to the variable.

```
else:
    return(vInput)
```

The keyword `return` tells Python to leave the function and *return* the inputted value to a variable.

Important: The `try...except...else` construct is used *inside* a `while True:` loop so that we can keep repeating the process until we get what we want.

Exercise 9.31.1 – Data Validation:

Before we write data validation functions to validate user entries, we will start with some simple *while True:* loops.

Exercise 1:

Using the code below (cut and paste), write a program that loops around a `while True:` loop until the user enters an integer;

```
while True:
    try:
        x = int(input("Please enter a number: "))
    except ValueError:
        print("Invalid entry, please try again. ")
        pass
    else:
        break
```

Notes:

1. This code snippet uses the keyword `break` to leave the `while True:` loop. This is because we are not in a function. The keyword `break` is used to leave a loop, but it is normally frowned upon as bad programming practice.
2. The correct indentation may be lost when you cut and paste the above code.

Exercise 2:

Amend your program to accept *two* numbers from the user (`userNum1 and userNum2`) and print their sum on screen as follow;

2 plus 3 is 5

Exercise 3:

1. Put the code from exercises 1 and 2 above to create a function called `inputInt()` that asks the user to enter an integer.
2. Use your new function `inputInt()` to get two numbers from the user; `userNum1` and `userNum2`.
3. You will need to call the function `inputInt()` function twice;
    a. `userNum1 = inputInt()`
    b. `userNum2 = inputInt()`
4. Because the `try...except...else` statement is now in a function, we will need to replace the keyword `break` with the keyword `return` to return the value;

    ```
    return vInput
    ```

5. Do not pass any parameters to the function `inputInt():`
6. Your `inputInt()` function should look something like the following;

```
#Function to get integer from user
def inputInt():
    while True:
        try:
            vInput = int(input("Please enter a number: "))
        except ValueError:
            print(vError)
            pass
        else:
            return(vInput)


#Call inputInt() function to get user's numbers
userNum1 = inputInt()
userNum2 = inputInt()

#Print addition on screen
print(str(userNum1) + " plus " + str(userNum2) + " is " + str(userNum1 + userNum2))
```

Exercise 4:

Add to parameters to your inputInt() function – *vQuestion* and *vError*;

1. *vQuestion* is the question to display on screen.
2. *vError* is the error message to display on screen when the user enters an incorrect data type.
3. Both parameters are passed to the function by the calling code;

```
userNum1 = inputInt(vQuestion, vError)
```

4. Both parameters are displayed on screen by the function.

9.32 Validating Yes/No Answers:

As you have probably noticed by now, computer systems ask lots of *yes/no* questions. In order to make sure we get the correct answer, we can write an `inputAnswer()` function that loops around a `while True:` loop until the user answers a correct response.

The simplest method of ensuring that we get a value that we want is to restrict the users' answer to a `tuple` of valid answers; Yes/No/Y/N, etc. We can then use an **`if...else`** statement to make sure that a valid answer was entered.

Exercise 9.32 Validating Yes/No Answers:

1. Copy your `inputInt()` function to a new function called `inputAnswer()`.
2. Declare a tuple of valid answers in the new function;
   a. `validAnswer = ("yes","y","no","n")`
3. Replace the `try...except...else` statement with a simple `if...else` statement as follow;

```
while True:
    vAnswer = input(vQuestion).lower()
    if vAnswer in validAnswer:
        return(vAnswer)
    else:
        print(vError)
        pass
```

4. **Notes:**
   a. `.lower()` is a Python function that converts data into lowercase. This makes it easier for us to check the user's entry.
   b. The `if vAnswer in validAnswer:` line checks the value the user entered against the list of values in the tuple `validAnswer`.
5. Call the `inputAnswer()` function to get a Yes/No answer from the user.
6. Using parameters, try different combinations of questions and error messages;
   a. Are the above details correct [Y/N]?
   b. Play Again [Y/N]?
7. **Feeling brave?** How about passing a tuple of valid answers to the function as a parameter;

```
validTuple = "male","female",m","f"
...
userAnswer = inputAnswer("Sex? ","Invalid response!!!",validTuple)
```

You could then use the one `inputAnswer()` function to get a valid answer to any question.

Note: Similar validation functions can be written for getting other data types from the user; dates, decimals, currency values, etc. Use Google to find working examples if needed.

9.33 Project 1 – Employee Details:

1. Using functions, write a program to accept the following employee details from the user and then print them on screen;
    a. **employeeID**: Integer value.
    b. **Name:** String that cannot be blank. Use an if...else statement to make sure something is entered;
    ```
    if vString == "" or vString == "":
    ```
    c. **Rate of Pay:** Float (decimal) value. The keyword `float` is used to accept decimal values; `vInput = float(input(vQuestion))`. The Rate of Pay will be used to calculate the employee's salary.
    d. **Hours Worked:** Float value. This is the number of hours the employee has worked this week. This will be used to calculate the employee's salary.

2. Use correct and appropriate variable names for each entry.

3. Based on the above entries, calculate the employee's salary;
    *Rate of Pay * Hours Worked*

4. Print the entered and calculated details on screen as follows;
    ```
    1001,John Lennon,9.50,39.5,375.25
    ```

5. Each value **must be** separated by a comma.

6. **IMPORTANT:** This program will be used again shortly to write employee details to a text file.

9.34 File Handling:

So far we have taken our information from the user via the screen. However, Python can get its information from many places; the internet, HTML forms, databases, computer systems, Excel workbooks, text files, etc.

Additionally, so far we have sent all of our output to the screen. However, Python can send its output to any type of file (as listed above)

Many major computer systems "share" data via *text files*. These are simple notepad style documents of rows and rows of data;

```
John,Lennon,Liverpool,England
Elvis,Presley,Memphis,USA
Daniel,O'Donnell,Donegal,Ireland
Johnny,Cash,Alabama,USA
```

This is an example of a text file containing famous male singers. The text file *singers.txt* could be *sent* from one computer system and *read* by another. In this way, large systems can *share* data extremely quickly and without the need to build interfaces.

9.34.1 Open a File to Write To:

Python uses the function `open()` to open a file. Depending on what we want to do with the file, the `open()` function takes a number of parameters;

*open("sample.txt", "**w**")*	This statement opens the text file sample.txt so that Python can **write** to it. If the file does not exist, Python will create it. If the does file already exists, Python will **overwrite it**.
*open("sample.txt", "**a**")*	This statement opens the text file sample.txt so that Python can **append** (add) to it. If the file does not exist, Python will create it. If the does file already exists, Python will **add to it**.
*open("sample.txt", "**r**")*	This statement opens the text file sample.txt so that Python can **read** it. Python cannot write to a file that was opened with the "r" parameter.

To open a file, we must assign the filename to a variable as follows;

```
outputFile = open("sample.txt", "w")
```

Whilst the variable name can be almost anything, it is good practice to name it `outputFile` or `outFile` as these variable names explain exactly what is going on in our program. Once a file is open we use the variable name when referring to it;

```
outputFile.write(“This is the first line of the new file sample.txt”)
```

So far all our output has gone to the screen with the `print()` function. If we want to write to an open file, we simply use the above syntax. Additionally, we can open multiple files for output (writing to) and for reading from at the same time – we simply give them different variable names (although this might get confusing).

Because we use a variable to store the name of the file, we can, if we needed to, ask the user for the name of the file with a simple `input` statement;

```
fileName = input(“Please enter the file name to write to?”)
outputFile = open(fileName, “w”)
outputFile.write(“Writing to... ” fileName)
...
outputFile.close()
```

When we are finished writing to an open file we must **close** it as above. We cannot write to a file that is not yet opened or that is closed.

RECAP: Here is a quick recap of the steps need to create a text file, write to it, and close it;

```
outFile = open(“sample.txt”, “w”)
outFile.write(“Hello, World!”)
outFile.write(“”)
outFile.write(“Another text line.\n\n”)
outFile.write(“The \n tells Python to print on the next line”)
outFile.close()
```

Exercise 9.34.1 – Write to a Text File:
Using the code snippet from above, write a program that creates the following text file;

```
Dear Customer,

As one of our most valued customers, we would like to reward you with
the gift of a brand-new car.

We thank you for your years of custom and wish you many happy years
of driving.

Yours sincerely,

The Management.
```

Notes: Don't forget the blank lines. It is more *Pythonic* to use the "\n" for moving the print onto the next line and for printing blank lines; ("\n\n").

9.34.2 Remember the Really Long String…?

Some time back we saw Python's `long string` data type. We saw that is started and ended with triple-single quotation marks (''') and that it kept its text layout. In case you have forgotten, here's a quick reminder;

```
screenDisplay = '''
This is a very long piece of text.
It goes on for ages.
Line after line.
It seems to never stop.
Is there no limit to the amount of data a Python string can hold.
I could literally be here all night...
'''
print(screenDisplay)
```

So, with Python's long string and file handling abilities in mind, we can write a Python program that creates an HTML page;

```
outFile = open("PythonHTMLpage.htm", "w")
outFile.write('''
<! DOCTYPE >
<html>
<head>
</head>
<body>
<p><b>Hello World!!!</b></p>
</body>
''')
outFile.close()
```

NOTES:

1. The above Python program creates a file called `PythonHTMLpage.htm`.
2. The new file will be created in your Python folder.
3. If the file `PythonHTMLpage.htm` already exists, Python will overwrite it.
4. If you have not covered *BITS Tutorial - Chapter 8: Web Design with HTML*, try writing a Python program that creates a Python program instead;

```
outFile = open("PythonFromPython.py", "w")
outFile.write('''
print("Hello World!!!")
''')
outFile.close()
```

You can then open your new Python program and run it.

After running the above program, find your new file `PythonHTMLpage.htm` in Windows *File Explorer* and double-click it. The HTML document will open in a web browser as an HTML document.

Exercise 9.34.2 – Create a Web Page from Within Python:
With the above in mind, write the following Python program;

1. Create an HTML page
2. Add a heading
3. Add 3 paragraphs
4. Use other HTML tags to develop your new HTML page.
5. Refer back to, *Chapter 8: Web Design with HTML* if required, or simply copy an existing HTML script.

9.34.3 Creating *Dynamic* Web Pages on the Fly:

The term "Creating dynamic web pages on the fly" simply means that we can use Python to create a web page based on information the user gives us. For example, we can ask the user for their name, favourite color, etc., and then use that information to build a web page that is personised to their given details.

This dynamic approach to creating web pages with Python is a little different than the previous examples and exercises. Previously we simply used a long string to write a *static* (unchanging) web page. If we ran our previous program ten times we would end up with just one web page which Python would continually overwrite each time we ran the program. However, by creating dynamic web pages we can ask the user for the name of the web page, and change the page details based on what the user entered.

Here's how to build dynamic web pages;

1. Ask the user a number of questions in relation to their web page;
    a. Please enter a name for your web page:
    b. Please enter your name:
    c. Please enter your favourite colour:
    d. What size heading would you like to use (1-6):
2. Based on the user input, we can now create a personalized web page using Python's *placeholders*.

9.35 Python Placeholders:

Placeholders are special characters that we can place in a long string. The placeholders then get replaced with user information when the program is run. A placeholder is defined with empty curly braces; **{}**. When Python encounters a placeholder, it places a related value into it;

```
longString = ("Hello {}, how are you today").format("John")
print(longString)
```

When the above code is run, the following is displayed on screen;

```
Hello John, how are you today
```

There is a lot going on here, so let’s take a detailed look at it;

1. The variable `longString` is assigned the value; *Hello {}, how are you today*.
2. When the code is run, the placeholder {} in `longString` is replaced with the name “John”.
3. **NB:** The entire string is in **brackets**.
4. The keyword `.format` is used to store the value for the placeholder.
5. Here’s the syntax;
    a. variable = (“... {} ...”).format(parameter).
    b. The placeholder is replaced by the value of the parameter.
6. We can have many placeholders in a long string once each placeholder has a value to be substituted in its place;

    ```
    longString = ("{} likes {} and {}, but not {}.").format("John","cats","dogs","rats")
    print(longString)
    ```

    Output; `John likes cats and dogs, but not rats.`

Exercise 9.35a – Using Placeholders:

1. Write a simple program that displays the following;

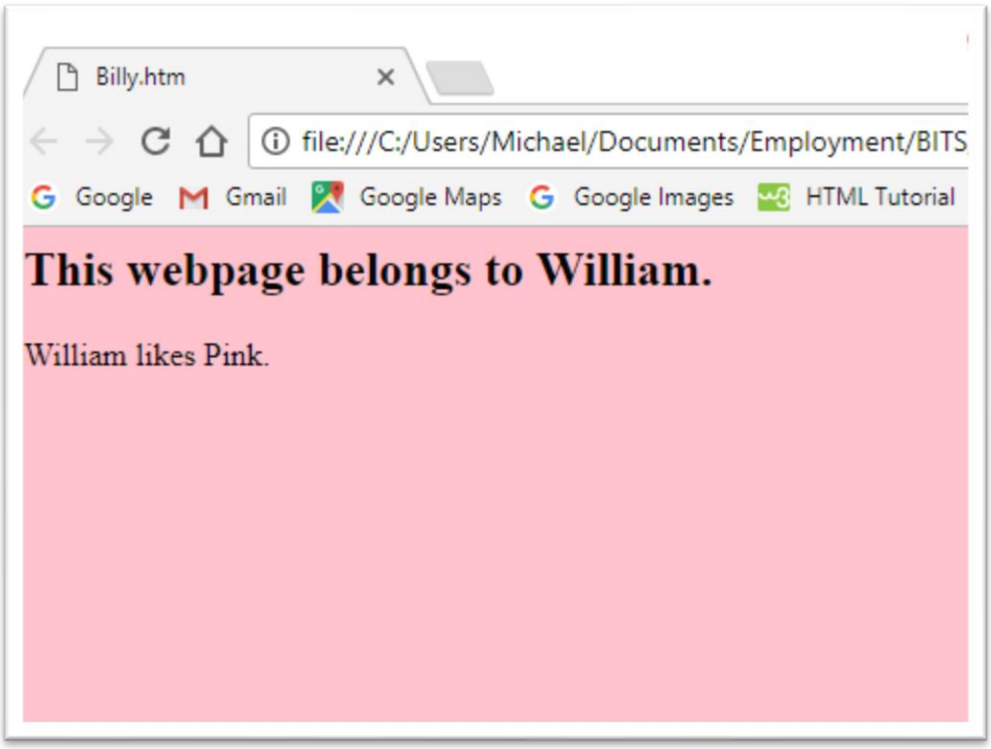
    John never did like Ringo or George, but he sure did love Paul.
2. Use placeholders for the values “John”, “Ringo”, ”George” and “Paul”.
3. Assign the string value to a variable and display it on screen.

Exercise 9.35b – Create a Dynamic Web Page with User Inputs:

Write the following program;

1. Ask the user for their name, favourite color and preferred heading size.
2. Using placeholders, create the web page `myHomePage.htm`.

3. Based on the user's input's, the web page should look something like the following;



This webpage belongs to William.

William likes Pink.

4. In this example, the user's name is William, his favourite colour is pink and he chose a heading size of 1.
5. Here's the HTML script for setting the background colour and heading size within a Python long string;

```
<body style="background-color:{};">
<h{}>This webpage belongs to {}.</h{}>
```

6. **Note:** You can set placeholders to use the user entries as often as you wish, once every placeholder has a value in the `.format()` function.

9.36 Numbering Placeholders:

The `.format()` list of values following the long string is a `tuple`. As we remember, a `tuple` is a data type that consists of a list of comma-separated values;

```
myFavouriteThings = ("Bats","Cats","Mats","Rats")
```

We can refer to values in a tuple based on their numerical position;

```
myFavouriteThings = ("Bats","Cats","Mats","Rats")

print(myFavouriteThings)
print(myFavouriteThings[1])
```

Output;

```
('Bats', 'Cats', 'Mats', 'Rats')
Cats
```

Remember, tuples are indexed from zero, so `myFavouriteThings[1]` displays "Cats"

With this in mind, let's return to our dynamic web page...

In our long string, we can use the numerical values of `{0}`, `{1}`, etc. instead of empty curly braces as placeholders. Each number refers to the position of the required value in the `.format()` tuple;

```
...''').format(userColour,headingSize,userName)
```

Therefore, we can refer to `userName` as {2} in our long string because `userName` is in index position 3 in our `.format()` tuple;

```
<body style="background-color:{0};">
<h{1}>This webpage belongs to {2}.</h{1}>

<p> {2} likes {0}. <p>
</body>
''').format(userColour,headingSize,userName)
```

As we can see from the above code snippet, we can use the placeholder `{2}` as often as we want and Python will insert the user's name each time.

Exercise 9.36 – More Dynamic Web Pages:

Based on the previous exercise, further develop the user's web page by;

1. Asking the user for a name for their web page document.
   a. Add the ".htm" extension to the user's input and open the file for writing.
2. Ask the user for a "b" or an "i" for emphasizing important text. If "b" use bold, if "i" use italics; <{3}>........</{3}>.
   a. The value "b" or "i" should be stored in a variable `userEmph` which can then be referenced in the `.format()` tuple.
3. Add more paragraphs of text to the web page.
4. Emphasize text in the new paragraphs with the `userEmph` value.
5. Add additional headers to the web page and use the user's heading size for each new header.

Some helpful tips;

Sample questions;

```
Please enter a name for your webpage: myHomePage
Please enter your name: John Lennon
What is your favourite colour: Yellow
What heading size would like on your website? 2
How would you like to emphasize your important text, bold or italic [B/I]? b
Webpage built. Double-click PythonHTMLpage.htm in your Python folder.
```

Sample long string;

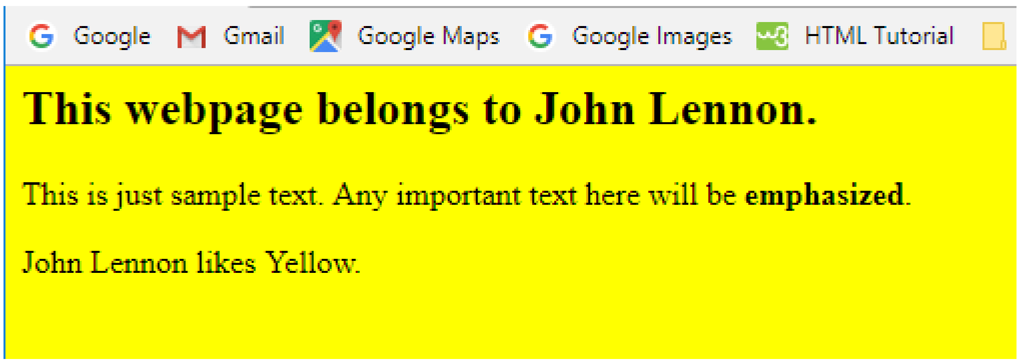
```
<h{1}>This webpage belongs to {2}.</h{1}>
<p>This is just sample text. Any important text here will be <{3}>emphasized</{3}>.<p>

<p> {2} likes {0}. <p>
```

Sample `.format()` tuple;

```
</body>
''').format(userColour,headingSize,userName,userEmph)
```

Sample dynamic web page;



This webpage belongs to John Lennon.

This is just sample text. Any important text here will be **emphasized**.

John Lennon likes Yellow.

9.37 Reading Files with Python:

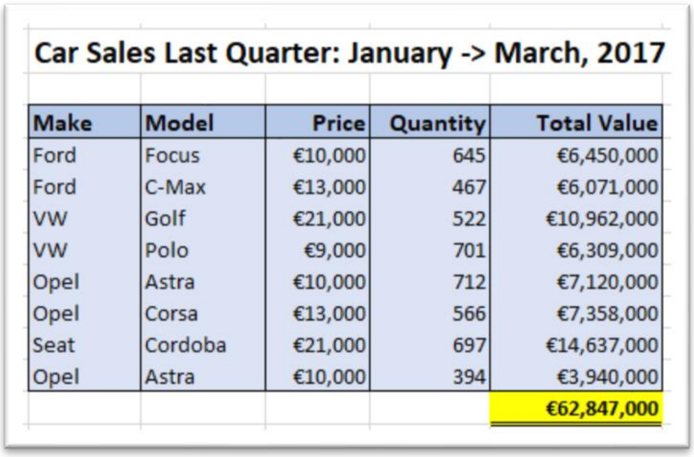
One of Python's main jobs on the internet and in business is to allow computer systems to communicate with each other. Django and HTML forms send the user's data Python which then processes the data and carries out some action (e.g. write it to a database, validate the entries, etc.). Python also works really well with Microsoft Excel workbooks (spreadsheets). Many companies use Excel as a management tool. They extract really important information such as sales figures, customer numbers, profits and losses, etc., and then import it into fancy spreadsheets for top management. With its abilities to work with the internet (Django and HTML forms), databases (Oracle, SQL, etc) and spreadsheets, Python is a powerful tool that can be used by programmers to gather information from one place and display/store it somewhere else.

An important feature of reading data in the **.csv** file. **CSV** stands for "**C**omma **S**eparated **V**alues" and is basically rows and rows of data, with the data in each row separated by commas;

```
Ford,Focus,"10,000",645
Ford,C-Max,"13,000",467
VW,Golf,"21,000",522
VW,Polo,"9,000",701
Opel,Astra,"10,000",712
Opel,Corsa,"13,000",566
Seat,Cordoba,"21,000",697
Opel,Astra,"10,000",394
```

(Look familiar? Remember our list of famous male singers.)

Whilst initially this data may seem confusing, once it is imported into Excel it all makes sense;



Car Sales Last Quarter: January -> March, 2017

Make	Model	Price	Quantity	Total Value
Ford	Focus	€10,000	645	€6,450,000
Ford	C-Max	€13,000	467	€6,071,000
VW	Golf	€21,000	522	€10,962,000
VW	Polo	€9,000	701	€6,309,000
Opel	Astra	€10,000	712	€7,120,000
Opel	Corsa	€13,000	566	€7,358,000
Seat	Cordoba	€21,000	697	€14,637,000
Opel	Astra	€10,000	394	€3,940,000
				€62,847,000

The above spreadsheet shows that the company sold a total of nearly €63million in car sales last quarter.

So, with the above in mind, let's start reading some .csv files...

Before we begin:

1. Use the following link to download the files *carSales.csv* and *readCarSales.txt* from the Python section of the Data File web page; Bits Tutorial website.
2. Copy/Move the files into *your* Python folder.
3. If the file *readCarSales.txt* opens in your browser instead of downloading, simply right-click it and select **Save As...** to save it locally.
4. **NB:** The file *readCarSales.txt* is a Python program that reads the *carSales.csv* file. Simply copy its contents into ILDE and save it as *readCarSales.py*.

9.38 Opening a .CSV File for Reading:

Below is the code of *readCarSales.py*. As you can see, there is a lot of new things happening here. Take a few minutes to read through it before continuing.

```
'''

Name:   readCarSales.py
Author: BITS Tutorial

Description:
This Python program reads the file CarSales.csv and then display each row of details on screen.
The data being read in is first placed in local variables.
The variable totalValue is calculated as (carPrice * qtySold).
The final print line is broken over a number of lines for clarity.
'''

#1. In order to work with .csv files we must first import the csv function from Python.
import csv


#2. Read each row of CarSales.csv.
with open('CarSales.csv') as carSales:

    #3. Place each row into the tuple "row".
    row = csv.reader(carSales)

    #4. Create the Python list "cell" and place each comma separated value in the current "row" into the list.
    for cell in row:

        #5. Place each value in the list "cell" into local variables.
        carMake     = cell[0]
        carModel    = cell[1]
        carPrice    = cell[2]
        qtySold     = cell[3]

        #6. Work on some local variable.
        totalValue = int(carPrice) * int(qtySold)

        #7. Print data to screen. This print line is broken over a number of lines for clarity.
        print("\n\n Make: "         + carMake  +
              "\n Model: "          + carModel +
              "\n Price: "          + carPrice +
              "\n Quantity Sold: "  + qtySold  +
              "\n Total Value: "    + str(totalValue))
```

To help explain what is going on in this program, the comments are printed and explained below;

1. In order to work with .csv files we must first import the csv function from Python.
Python has many system functions that we can import into our code to use. We use the keyword `import` to import a system function. Because this program works with .csv files, we must import the system function `csv` as follows; `import csv`

2. Read each row of CarSales.csv.
The line; `with open('CarSales.csv') as carSales:` opens the file *CarSales.csv* for reading and places its contents into the long string variable `carSales`.

3. Place each row into the tuple "row".
The line; `row = csv.reader(carSales)` places each row in the .csv file into a Python list variable called `row`. This allows us to use each row of the .csv file one at a time. The function `csv.reader()` is a Python function (that we imported from Python with `import csv`) used to read data in from data files.

4. Create the Python list "cell" and place each comma separated value in the current "row" into the list.
The line; `for cell in row:` uses a `for` loop to read through each comma separated value in the current row. Every value is placed into the list variable `cell`. Once we have the data in a list variable, we can now do what we want with it.

5. Place each value in the list "cell" into local variables.
This code places values from the list variable `cell` into local variables for us to work with.

Note: At this stage of the process, we have opened an external file, read in the data, and placed it in local variables so that we can use it. From here on the program can be read as a normal Python program.

6. Work on some local variable.
This is a straight forward variable assign statement.

7. Print data to screen. This print line is broken over a number of lines for clarity.
This code displays the values in the local variables (that we gleaned from the.csv file) on screen. It is broken over multiple lines for clarity.

8. Get next row from the .csv file:
At this point in the processing, the program loops and reads in the next row of data for processing.

Note: The key to working with .csv files is to get the comma-separated values into variables so that we can work with them. For example, once we have values such as `carPrice` and `qtySold` in local variables, we can use them find the `totalValue` of sale this quarter.

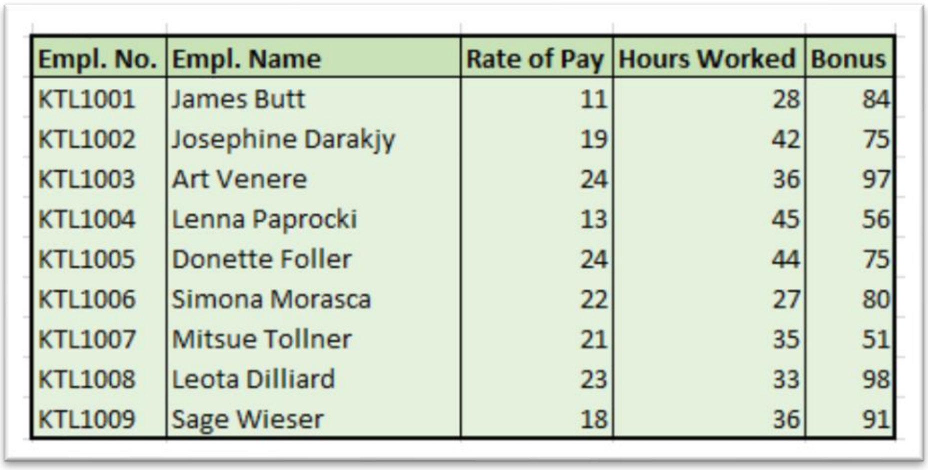
Exercise 9.38 – Read employees.csv:

Use the following link to go to the BITS Tutorial website and then download the file *employees500.csv* from the Python section; BITS Tutorial website.

The file *employees500.csv* contains the following details for 500 pretend employees;

1. Employee Number	*(e.g. KTL1001)*
2. Employee Name	
3. Hourly Rate of Pay	*(e.g. 12)*
4. Hours Worked per Week	*(e.g. 34)*
5. Weekly Bonus	*(e.g. 51)*

Here's a sample of what the data would look like in a formatted Excel workbook;



Empl. No.	Empl. Name	Rate of Pay	Hours Worked	Bonus
KTL1001	James Butt	11	28	84
KTL1002	Josephine Darakjy	19	42	75
KTL1003	Art Venere	24	36	97
KTL1004	Lenna Paprocki	13	45	56
KTL1005	Donette Foller	24	44	75
KTL1006	Simona Morasca	22	27	80
KTL1007	Mitsue Tollner	21	35	51
KTL1008	Leota Dilliard	23	33	98
KTL1009	Sage Wieser	18	36	91

Note: *Raw data (lots of lots of rows of data in .csv files) rarely contain row headers, labels, currency symbols, etc. This is because, for example, it would be hard to multiply the words "Rate of Pay" by the words "Hours Worked". Labels, currency symbols, etc. would just get in the way of the data that needs to be processed. Headers, labels, etc would normally be added at print time, but rarely stored.*

Exercise:

1. Using *employees500.csv* and *readCarSales.py*, write a program to calculate the following;
   a) Cost of the employees' wages for the week;
      ```
      empWages = (rateOfPay * hoursWorked)
      ```
   b) Total cost of wages for the week;
      ```
      totalWages = totalWages + empWages
      ```
   c) Total cost of bonuses paid this week;
      ```
      totalBonus = totalBonus + emplBonus
      ```
   d) Total amount spent by the company this week;
      ```
      totalSpend = totalWages + totalBonus
      ```
2. **NB:** Do not print anything *inside* the reading of *employees500.csv* as this would print 500 lines on the screen. Instead, just print the totals;
   ```
   Wages Total:  $341,735.00
   Bonus Total: $37,813.00
   Total Cost:  $379,548.00
   ```

3. **Printing Currency:** Python uses a very strange way to print currency values. It inherited this strange beast from C (the programming language that Python is written in). Here is the code for displaying a float data type as a currency;

   ```
   print("${:,.2f}".format(totalWages))
   ```

   This code prints the variable `totalWages` as; `$341,735.00`

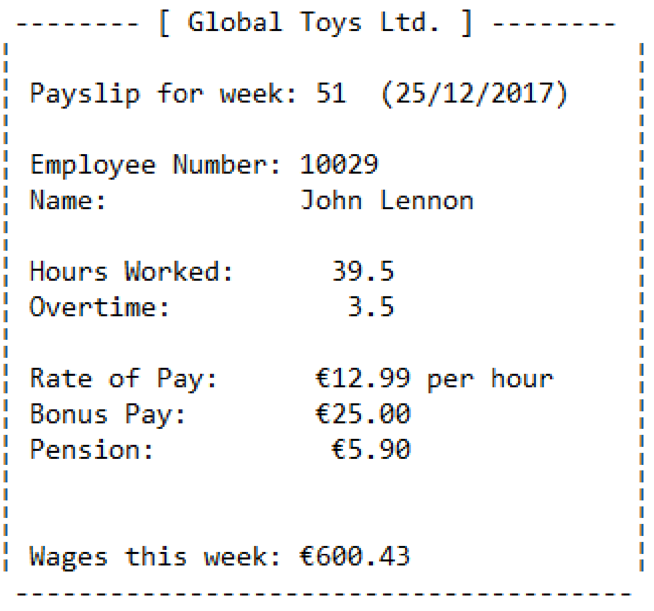
   As you can see, if you look closely enough, it is obviously a placeholder (with the curly braces and the `.format()` function). Anyway, we could spend the next two hours explaining what each part means and why they are there, but let's just accept it and move on...

4. The printing of the totals is a normal print statement which includes the above syntax;

   ```
   print("\n\nWages Total: ", "${:,.2f}".format(totalWages),
         "\nBonus Total:", "${:,.2f}".format(totalBonus),
         "\nTotal Cost: ", "${:,.2f}".format(totalSpend))
   ```

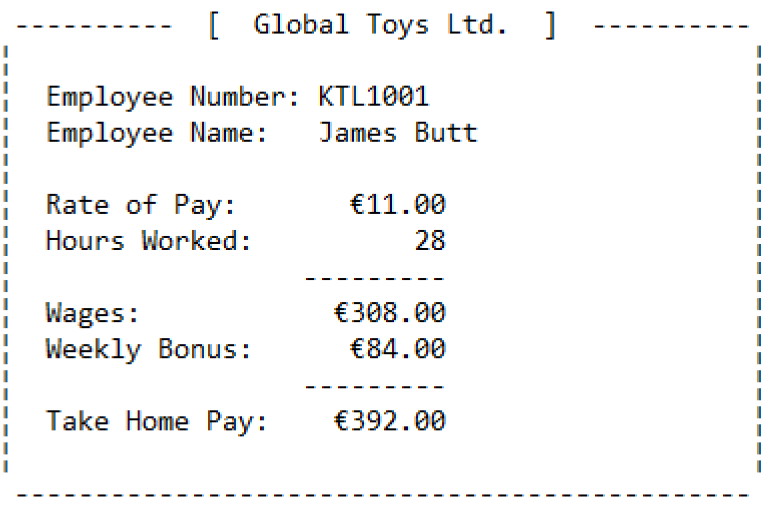
9.39 Payroll Revisited:

Remember our payroll program from earlier...?



```
 -------- [ Global Toys Ltd. ] --------
|                                        |
| Payslip for week: 51  (25/12/2017)     |
|                                        |
| Employee Number: 10029                 |
| Name:            John Lennon           |
|                                        |
| Hours Worked:      39.5                |
| Overtime:           3.5                |
|                                        |
| Rate of Pay:      €12.99 per hour      |
| Bonus Pay:        €25.00               |
| Pension:           €5.90               |
|                                        |
|                                        |
| Wages this week: €600.43               |
 ---------------------------------------
```

It is now time to revisit it, but this time the input will be coming from the *employees500.csv* file instead of the screen. Because our raw data is a little different than before, here's the new layout of the payslip;



```
----------  [  Global Toys Ltd.  ]  ----------
|                                             |
|  Employee Number: KTL1001                   |
|  Employee Name:   James Butt                |
|                                             |
|  Rate of Pay:       €11.00                  |
|  Hours Worked:          28                  |
|                  ---------                  |
|  Wages:            €308.00                  |
|  Weekly Bonus:      €84.00                  |
|                  ---------                  |
|  Take Home Pay:    €392.00                  |
|                                             |
----------------------------------------------
```

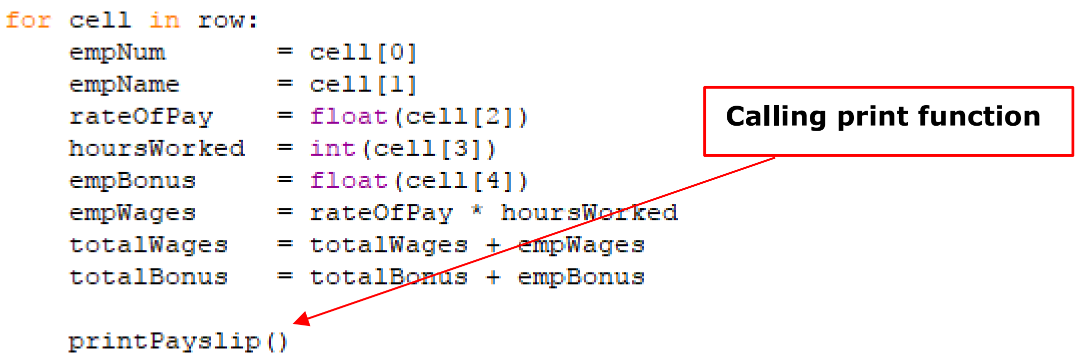
As you can see, most of the details on the payslip come straight from the *employees500.csv* file. The only figure that does not come from *employees500.csv* is *Wages this Week* (empWages + empBonus).

Before we begin: There are too many employees in our source file *employees500.csv* at present. Open it in Notepad and delete from employee **KTL011** to the end of the file, leaving only the first ten employees in the file. (You can rename the modified file to *employees10.csv* if you wish, but remember to change your program to reflect this change).

Exercise 9.39 – Payroll Program Revisited:

Write a program to create payslips for employees listed in a .csv file.

1. The new version of our payroll program will read a .csv file of employee details.
2. It will then call a `printPayslip()` function directly after the .csv values have been assigned to local variables;



```
for cell in row:
    empNum        = cell[0]
    empName       = cell[1]
    rateOfPay     = float(cell[2])
    hoursWorked   = int(cell[3])
    empBonus      = float(cell[4])
    empWages      = rateOfPay * hoursWorked
    totalWages    = totalWages + empWages
    totalBonus    = totalBonus + empBonus

    printPayslip()
```

3. The `printPayslip()` function will print a payslip for the current employee;

```
def printPayslip():
    outFile.write("\n\n ----------  [  Global Toys Ltd.  ]  ---------- \n")
    outFile.write("¦                                              ¦\n")
    outFile.write("¦  Employee Number: " + empNum + "                   ¦\n")
    outFile.write("¦  Employee Name:   " + empName.ljust(20) + "       ¦\n")
```
etc….

4. **Note:** As most payslips are printed on pre-printed stationery, do not spend too much time trying to align the border around the payslips.

5. The program will also write to a *payrollReport.txt* file which will list details of the payroll run as follows;

```
KTL1001  James Butt        €11.00  28  €308.00  €84.00  €392.00
KTL1002  Josephine Darakjy €19.00  42  €798.00  €75.00  €873.00
KTL1003  Art Venere        €24.00  36  €864.00  €97.00  €961.00
KTL1004  Lenna Paprocki    €13.00  45  €585.00  €56.00  €641.00
KTL1005  Donette Foller    €24.00  44  €1,056.00  €75.00  €1,131.00
KTL1006  Simona Morasca    €22.00  27  €594.00  €80.00  €674.00
KTL1007  Mitsue Tollner    €21.00  35  €735.00  €51.00  €786.00
KTL1008  Leota Dilliard    €23.00  33  €759.00  €98.00  €857.00
KTL1009  Sage Wieser       €18.00  36  €648.00  €91.00  €739.00
KTL1010  Kris Marrier      €22.00  40  €880.00  €66.00  €946.00
```

a) To write to a second file, simply use a second `open` statement with the filename `payrollReport.txt`;

```
outFile = open("payslips.txt", "a")
reportFile = open("payrollReport.txt", "w")
```

b) The *payrollReport.txt* file should be opened with the "`w`" parameter so as to overwrite any existing Payroll Reports.

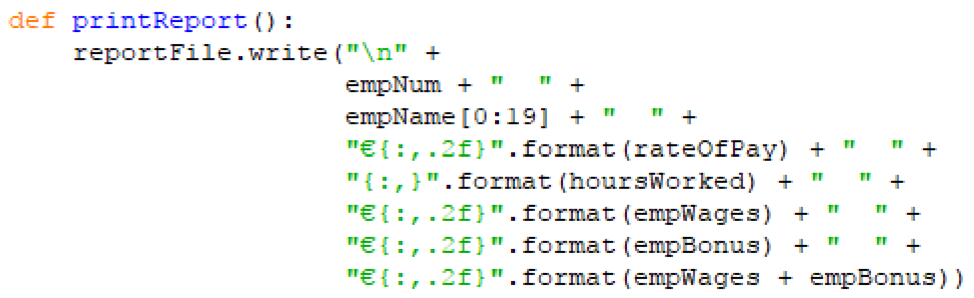
6. The `printReport()` function will be called straight after the `printPayslip()` function;

```
for cell in row:
    empNum        = cell[0]
    empName       = cell[1]
    rateOfPay     = float(cell[2])
    hoursWorked   = int(cell[3])
    empBonus      = float(cell[4])
    empWages      = rateOfPay * hoursWorked
    totalWages    = totalWages + empWages
    totalBonus    = totalBonus + empBonus

    printPayslip()
    printReport()
```

Calling 2nd print function

7. The `printReport()` function will simply print current employee details to the *payrollReport.txt* file;



```
def printReport():
    reportFile.write("\n" +
                     empNum + "   " +
                     empName[0:19] + "   " +
                     "€{:,.2f}".format(rateOfPay) + "   " +
                     "{:,}".format(hoursWorked) + "   " +
                     "€{:,.2f}".format(empWages) + "   " +
                     "€{:,.2f}".format(empBonus) + "   " +
                     "€{:,.2f}".format(empWages + empBonus))
```

8. Do not forget to close all open files at the end of your program. Unclosed files will appear empty in File Explorer because Python still has control of them.

9. **Notes:**
   a) The string `empName` can be of any length (because the number of characters in a name can be of any length). Use the code `empName[0:19]` to only print the first 20 characters of the `empName`.
   b) Don't forget to **import** the **csv** system function.
   c) Remember, "**/n**" is used to move the print on to a new line, and **end = ""** is used to keep the print on the current line.

10. **Feeling brave?** Add report totals to the file *payrollReport.txt*.

9.40 System Functions:

In the above section, we imported the Python system function **csv** so that we could work with *.csv* files in our Python programs.

Python has many such system functions that we can import and use just like the functions we write ourselves. Whilst the list of Python system functions is enormous (and quite often a bit technical), here are some simple and useful ones;

9.40.1 Time:

The Python system function `time` can be used to pause a program for a specified number of second;

```
import time

print("This program... ", end="")
time.sleep(2)
print("pauses for...", end="")
time.sleep(2)
print("two seconds.")
time.sleep(2)
```

Exercise 9.40.1 Pausing a Program:

Write and run the above program.

Remember: The code **end=""** tells Python not to insert a new line at the end of the print statement.

9.40.2 Windows Sounds:

We can play any of the Windows sounds from within Python by importing the `winsound` function;

```
    import winsound

    winsound.MessageBeep(1)
```

This code snippet sounds a simple Windows beep. This can be useful when displaying error messages in validation functions.

The list of available *winsounds* is extensive. Click here for a list of all available Windows sound files.

The following code snippet plays a number of Windows sounds after one-second pauses;

```
# Import the Python winsound and time functions...
import winsound, time


# Assign the sound files to local variables...
sound1 = "c:/Windows/Media/chimes.wav"
sound2 = "c:/Windows/Media/Alarm08.wav"

#Play the Windows chimes.wav sound file...
winsound.PlaySound(sound1, winsound.SND_FILENAME|winsound.SND_ASYNC)

# Pause 1 second before playing the Windows Alarm08.wav file...
time.sleep(1)
winsound.PlaySound(sound2, winsound.SND_FILENAME|winsound.SND_ASYNC)

# Pause 1 second before playing the Windows Sysytem Exit .wav file...
time.sleep(1)
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)
```

You can also create your own sounds with the `winsound.Beep(frequency,duration)` function. Try these few examples (you might want to plug in your headphones before you drive your teacher mad!!);

```
#The parameters for winsound.Beep are frequency (pitch) and duration.
winsound.Beep(327,500)
winsound.Beep(1327,500)
winsound.Beep(2327,500)
```

The parameters for the `winsound.Beep()` function are *frequency* and *duration*;

1. *Frequency* relates to the pitch or tone of the beep and can have a value from 37 to 32767.
2. *Duration* is measured in milliseconds.

Exercise 9.40.2 – Playing Windows Sounds from a Python Program:

1. Try a few different combinations of frequency and duration with the `winsound.Beep()` function.
2. Use the above link to available winsounds to play a number of Windows sound files. Locate a sound file that you think would be useful when writing Python programs and make a note of it (Notepad).

9.40.3 DateTime:

The Python system function `datetime` allows us to import the current time and date into a program;

```
#Read and store the current date and time
import datetime
```

If we print `datetime` Python display the name of the program that the datetime function was called from;

```
<module 'datetime' from 'C:\\Python36-32\\lib\\datetime.py'>
```

Obviously, this is of little use to us in our programs. However, we can extract more meaningful information about today's date and time from the `datetime` function as follows;

```
systemDate = datetime.datetime.now()
```

This statement assigns the current date and time to a local variable called `systemDate`. We can now display more meaningful information;

```
print(systemDate)
```

Output; `2017-08-31 12:35:03.045150`

Okay, whilst we can now see the current date and time, this information is still a little unhelpful. However, we can now start extracting meaning information from the variable `systemDate`;

```
    todaysDate       = systemDate.strftime("%d/%m/%Y")
    todaysTime       = systemDate.strftime("%H:%M:%S")
    todaysDayOfMonth = systemDate.strftime("%d")
    todaysMonth      = systemDate.strftime("%m")
    todaysYear       = systemDate.strftime("%Y")
```

Important: *The exact syntax of the above assign statements is required to properly extract the relevant details.*

The above statements extract the relevant information into local variables. We can now print meaningful data;

```
Today's date:   31/08/2017
Current time:   12:35:03
Day of month:   31
Current month:  08
Current year:   2017
```

Because we have now assigned the system date and time to local variables, we can now use them in a program (or write them to .txt reports);

```
Report Date:  31/08/2017

Payroll run commenced at 12:54:07 on the 31 day of month 08, 2017
```

We could also create a Python list of months and display the name of today's month;

```
months = ("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sept","Oct","Nov","Dec")
printMonth = int(todaysMonth) - 1

print("\nReport Date:  " + todaysDate)
print("Payroll run commenced at " +
      todaysTime         + " on " +
      months[printMonth] + " "    +
      todaysDayOfMonth   + ","    +
      todaysYear + ".")
```

**Remember, tuple indices (indexes) start at zero.*

Exercise 9.40.3 Display Today's Date:

Use the code snippets above to display today's date and time in different formats.

Note: Use this link for more information about working with dates and time in Python.

9.41 Importing Our Own Functions:

As well as importing Python system functions, we can also import our own functions from one program to another. For example, we can write a program of validation functions and then import them (one by one, or all together) into another program. This is a common programming strategy. It is quite common for teams of programmers to share a single program that consists of many commonly used functions. Then, if a new shared function is required, it can be added to the shared program once and all developers can use it.

9.41.1 Shared Functions Example:

Program One: sharedFunctions.py (extract):

```
# Name:   sharedFunctions.py - contains many shared functions
# Author: BITS Tutorial

def inputInt(vQuestion, vError):
    while True:
        try:
            vInput = int(input(vQuestion))
        except ValueError:
            print(vError)
            pass
        else:
            return(vInput)
```

Program Two: getUsersAge.py:

```
# Name:   getUsersAge - test program for importing shared functions
# Author: BITS Tutorial

#Call inputInt() function from sharedFunctions.py
from sharedFunctions import inputInt

#Use inputInt() functions as normal...
myAge = inputInt("\nPlease enter your age: ", "Please enter a valid age!!!")

#Some sample code...
print("You said you were " + str(myAge) + ".")
```

The program *getUsersAge.py* above imports the function `inputInt()` from the program *sharedFunctions.py*. The program *getUsersAge.py* can use the shared function just as if it were written locally.

9.41.2 Syntax of Calling Functions From Another Python Program;

1. To call a single function we must name the program and the function;

   ```
   from sharedFunctions import inputInt
   ```

   i) Notes:

   (1) The *.py* extension is not required on the program name being called.

   (2) We do not use the function brackets () when calling the function.

   (3) We do not pass any parameters when calling the function.

2. To call multiple functions from another Python program;

   ```
   from sharedFunctions import inputInt, inputFloat
   ```

   i) Notes:

   (1) Each function is separated by a comma

3. To call all functions from another Python program;

   ```
   from sharedFunctions import *
   ```

Exercise 9.41.2 Shared Functions:
Write a program that calls functions from another program to validate user input;

1. Use the above code snippets and your existing programs to complete this task.

2. **Note:** If you get an error message saying that Python cannot find the called program, you will need to check a few things;
   a. Make sure that the program and the function being called exist (spelling).
   b. Make sure the called program is in the same folder as the calling program.
   c. If the error continues it may be due to something called the PYTHONPATH environment variable not being setup properly. If this is the case, simply skip this exercise.

9.42 Object Orientated Programming:

Object Orientated Programming (OOP) is really, really confusing. It is so confusing that it will take you a long time to get your head around it - and even then, you won't quite be sure that you understand it fully.

So, to explain OOP we will need to keep things very, very simple...

An *object* (in programming terms) is basically a piece of code that does something. We have seen many objects so far in this tutorial; a function is an object because it is just a piece of code that does something, a Python system function is an object because it is just a piece of code that does something. We could get silly and say that every piece of code is an object because it is just a piece of code that does something. Whilst we would be technically correct in so saying, we would kind of be missing the true intent of *objects* in OOP.

So, now that we agree that an object is just a piece of code that does something (like a function), we can take a closer look at OOP.

OOP is about writing code that will create an object (piece of code) when the program is run, but the actual object does not exist in the program.

Help!!!!!

Okay, let's start again.

If we decide to go to the shop at lunchtime and buy some chips, we know that we will eat the chips when we buy them. But right now, we haven't gone to the shop yet and, as such, we don't have any chips to eat. We know that we will enjoy the chips at some point in the future (lunchtime), but for now, we don't actually have any chips.

Similarly, OOP is about writing code that will create objects when the program is run, but which are not actually coded in the program. We know that when the program is run it will create the object and that the object will do exactly what we tell it to, but if we looked in the program we would not find the object.

Told you OOP was confusing!!!

Okay, let's look at a few real-life examples of OOP to simplify things further.

9.42.1 OOP - Scenario One – New Customer System:

Congratulations, your company has just signed a new contract with a new customer and your boss wants you to write the following programs for the new IT system;

1. A customer setup program for creating and entering customer details.
2. A supplier setup program for creating and entering supplier details.
3. An employee setup program for creating and entering employee details.

4. A product setup program for creating and entering product details.
5. Etc..., we get the idea.

As we can see, all the programs listed above are basically the same except for the type of data being saved (customer, supplier, etc). All of these programs would store a code, a name, an address, an identifier of some sort (telephone number or product-id), etc. So, it would be a complete waste of time and space if we wrote 4 different programs that did pretty much the same thing. Agreed?

9.42.2 OOP – Scenario Two – New Gaming Sensation:

Congratulations, your company has just signed a deal to write the new version of Doom and your boss wants you to write the characters. Awesome!!!

In developing the characters you must capture the following information for each of the twenty character types;

1. Weapon
2. Costume
3. Strength
4. Abilities
5. Ammo
6. Food
7. Etc., we get the idea

Do we really want to sit there for weeks writing 20 different character setup routines when *we could write just one and pass it a value*?

This is where OOP comes in.

With OOP, we could write one program that creates any number of characters for our game. We could ask the user which character we are to create and our program could the rest. If the user has chosen Hero1, we can then create the character Hero1 and give him/her all the attributes (bits) associated with that character type.

Similarly, in scenario one, if the user tells the program what type of entity (customer, supplier, product etc.) is being created, then the program can create that entity. In this way, we do not need to write lots of setup programs

Hopefully, by now, OOP is a little clearer for you. It's about saving code. It's about saving time. It's about saving processing time and thus making our computer systems and apps faster, better and cheaper.

9.42.3 Writing OOP:

Here is the definition of an OOP *class* in Python;

```
class Setup:
    # Initiating function...
    def __init__(self, enType):
      self.ID           = ""
      self.name         = ""
      self.description = ""
      self.email        = ""
      self.contact      = ""
      self.rateOfPay    = 0.00
      self.startDate    = ""

    #-- Function to allow data entry...
    def inputDetails(self, file):
        self.ID          = file
        self.name        = input("  Enter " + file + "'s name : ")
        self.description = input("  Enter " + file + "'s address : ")
        self.email       = input("  Enter " + file + "'s email : ")
        self.contact     = input("  Enter " + file + "'s contact : ")
        self.description = self.description.replace(",", " ")
        self.startDate   = input("  Enter " + file + "'s start date : ")
```

As you can see, it is quite similar to the definition of a function.

In Object Orientated Programming;

1. A `class` is simply a container for `methods`.
2. A `method` is basically a `function`.

So, the methods `_init_()` and `inputDetails()` are just functions that have been defined in a class called `Setup`. Based on our knowledge of functions, we should be able to understand and make sense of the `method inputDetails()`.

As we can see, the `inputDetails()` method is assigning user entries to a list of variables that are attached to a funny thing called `self`. This `self` is also the first parameter of the method;

```
def inputDetails(self, file):
```

In OOP programming, the `self` refers to the second parameter being passed into the method – in this case, `file`. This basically tells Python to replace the word `self` with the *value* of the parameter `file` when the program is run. So, if the method is passed the parameter *customer*, it will assign the screen entries to customer.ID, customer.name, customer.description, etc. If the program is run a second time and this time the method is passed the parameter *supplier*, it will assign the screen entries to supplier.ID, supplier.name, supplier.description, etc.

NB: The first parameter of all defined methods **must be** `self`. However, this parameter does not get passed to the methods by a calling program.

To understand the `self.` a little better, we first need to look at the code from the calling program. Here's the relevant code snippet;

```
fileType = input("Type of file to create : ").lower()

file = Setup(fileType)

file.inputDetails(fileType)
```

Line 1 asks the user for the type of file to create (customer, supplier, etc.).

Line 2 uses the class `Setup` to create an instance (a temporary snapshot) of the `fileType`. If the user entered "customer", the program dynamically creates a customer instance. If supplier, then an instance of a supplier object is created, etc. Hence the term "Object Orientated Programming" – different objects are created (in memory) based on the user's input.

Line 3 calls the `inputDetails()` method from the `Setup` class with the current instance (e.g. customer). If the user has entered "customer" on line 1, then this method asks the user to enter *customer* details. The `self.` code in the `inputDetails()` method is replaced with *customer*, and *customer* data is entered by the user. If we could see the instance of the customer object it would probably look like this;

```
def inputDetails(customer):
    customer.ID          = file
    customer.name        = input("  Enter " + file + "'s name : ")
    customer.description = input("  Enter " + file + "'s address : ")
    customer.email       = input("  Enter " + file + "'s email : ")
    customer.contact     = input("  Enter " + file + "'s contact : ")
    customer.description = customer.description.replace(",", " ")
    customer.startDate   = input("  Enter " + file + "'s start date : ")
```

However, this instance only exists when the program is running. Once it stops, the instance is cleared from memory. If the user then ran the program to set up a supplier, the program would create a supplier instance in memory instead;

```
def inputDetails(supplier):
    supplier.ID          = file
    supplier.name        = input("  Enter " + file + "'s name : ")
    supplier.description = input("  Enter " + file + "'s address : ")
    supplier.email       = input("  Enter " + file + "'s email : ")
    supplier.contact     = input("  Enter " + file + "'s contact : ")
    supplier.description = supplier.description.replace(",", " ")
    supplier.startDate   = input("  Enter " + file + "'s start date : ")
```

Which would again cease to exist when the program stopped.

Note: To further develop the class `Setup`, we could add a *print* method that would print the current instance;

```
def writeDetails(self, file):
    file.write(self.name        + "," +
               self.description + "," +
               self.email       + "," +
               self.contact)
    if self.ID == "employee":
        file.write("," + self.rateOfPay)
    file.write("," + self.startDate + "\n")
```

Methods can be called by a calling program in the same way that we called functions from another program earlier. However, the norm is to call the entire class into your program;

```
from ClassDefinitions import Setup
```

Where *ClassDefinitionns.py* is a separate program containing the class.

The _init_ Method:

The first method in the above class is called the `_init_()` method. It is a special case method. It *must* always be present in a class definition and it must always be the first method defined in a class. Put simply, it is the method that initiates (sets up) all the data that is available to be used by the rest of the methods in a class. It can be considered the blueprint of the objects that are going to be created when the program is running.

It is well known that Object Orientated Programming is very confusing. It is also well known that a picture paints a thousand words...

So let's have a look. Use this link to go to the BITS Tutorial website and then download the files *classDefinitions.txt* and *weAreCarpets.txt*. These files are Python programs that use the OOP techniques outlined above. Rename the files to have *.py* extensions to make them Python programs

Important: Do not worry if you cannot follow OOP. There is plenty of time to learn when you get to college...

Exercise 9.42.3 – Object Orientated Programming:

3. Download the files *classDefinitions.txt* and *weAreCarpets.txt* from the BITS Tutorial website.
4. Rename the files to Python programs (.py extension).
5. Open both programs in IDLE (the Python editor) and follow the logic. Try to figure out what is happening, and when.
6. The programs write to .txt files, but in a real-life environment, the output would be sent to spreadsheets, databases, websites, the cloud, etc., with the input data coming from the same.
7. **Feeling brave?** Fancy writing an OOP program to setup a database text file of characters for a new release of a super cool video game. Here's how to do it;
    a. Identify want information you will need to store on each character;
        i. Name
        ii. Type – Wizard, warrior, maiden, friar, king, price, baddie
        iii. Strength – Assign a level of strength to each character type
        iv. Weapon – Wand, sword, gun, zombie-killing-laser-blaster
        v. Ammo – as iii (integer values from 1 to 100)
        vi. Food – as iii
        vii. Energy – as iii
        viii. Any other gaming setting you might need
    b. Your program should create a text file for each character type. You should end up with a text file of wizards, warriors, maidens, monsters, etc. In a real-life environment, these files would be written to database records (similar to text files). The game would then read the files to create the gaming environment.
8. **Feeling super brave?** *Pygame* is set of Python modules designed for writing video games in Python. It's completely free to download and comes with great tutorials and examples. It can be used with Django to create fully functioning gaming environments that can be released on the web. Feel like a career in gaming...?

9.43 Other Important Python Features:

Python has many more features than those covered in this introductory tutorial. The complete list of all available features, functions, modules, etc would be too large to include here. However, a simple rule for finding Python features is to type "Python" followed by your query into Google. (For example, "Python convert a tuple to a list".)

Not all websites offer the best solution. Never simply accept the first site you find as gospel. Try to find some tried and trusted sites. Only experience can guide you.

With that said, here is a list of two main Python features that this tutorial did not cover.

9.43.1 The `dict` Data Type:

In Python, `dict` in a data type. As its name suggests, the `dict` data type is used as a dictionary. In a normal English dictionary, we can look up a word and see its definition beside it. Dictionaries in Python work the very same way. For example, supermarket computer systems can use Python dictionaries to store product prices;

```
store_prices  = {"Apples": 3.50, "Bananas": 2.99, "Carrots": .99, "Grapes": 2.47}
```

As we can see, the `dict` data type is made up a list of paired items, separated by a comma, *inside* curly braces;

myDictionary = {key: value, key: value, etc..}

The *key* part of a `dict` pair is the thing we look up and the *value* part is the thing we want to find out. For example, in an ordinary English dictionary, the *key* would be the word we are looking up and the *value* would be the definition associated with the word.

The `dict` data type can be used as an actual dictionary;

```
french  = {"Dog":"Chien", "Horse":"Cheval",  "Cow":"Vache"}
spanish = {"Dog":"Perro", "Horse":"Caballo", "Cow":"Vaca"}

user = input("Enter nationality: ")

if user == "French":
    print(french["Dog"])
elif user == "Spanish":
    print(spanish["Dog"])
else:
    print("Sorry, language not supported.")
```

Exercise 9.43.1 Using `dict`– Supermarket Lookup:

Write a program to print (on screen) the price associated with a product;

1. Create a `dict` of 5 products (product name and price)
2. Ask the user to enter a product name.
3. Print the product's price on screen in a sentence as follows;

```
Today's price for bananas is €1.49.
```

4. **Feeling brave?** Write a function to validate the user entry against the product names in the `dict`. This function would be similar to the Yes/No validation function.

9.43.2 String Manipulation:

Python string variables are very powerful. The *long string* can store whole web pages, books, etc. Theoretically, the maximum storage capacity of the Python long string is dependent on the size of your hard disk and available memory. This could be a storage capacity of over 63 gigabytes of information in one string (that's nearly 16,000 bibles worth of data).

The data in Python strings can be searched, split, replaced, parsed, overwritten character by character, reversed, converted to ASCII, hex, octal, uppercase, lowercase, lists, tuples - the list goes on.

The `.split()` Function:

The `split()` function separates a comma separated string value into a Python list;

```
#-- Using the split() funtion to separate a list into individual variables...
colourList = "blue, red, green, black"
colour1, colour2, colour3, colour4 = colourList.split(",")
print("List of colours: ", colourList)
print("Variables       : ", colour1, colour2, colour3, colour4)
```

Print Part of a String:

Sections of a Python strings can be printed (or assigned to a variable) as follow;

```
#-- Print parts of a string (indexed from 0)...
x = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
y = x[0:4]
print(y)
```

The `.find()` Function:

The `find()` function returns the integer position of a string segment;

```
#-- Find value in string (indexed from 0)...
x = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
y = x.find("J")
print(y)
```

The `.count()` Function:

The count() function returns the number of times a specified segment occurs in a string;

```
#-- Counts the occurances of KML in string...
x = "ABCDEFGHIJKLMNOPQRSTUVWXYZKLMKLMKLM"
print(x.count("KLM"))
```

Note: The string `x` has an added "KLMKLM" in this example.

The `.lower()` and `.upper()` Functions:

The `.lower()` functions returns the lowercase version of a string;

```
x = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print(x.lower())
```

Combining String Manipulation Functions:

Any and all of the above string functions can be used together;

```
x = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print(x[0:4].lower())
```

Exercise 9.43.2 - String Manipulation:

1. Write a program that contains a string variable (`x`).
2. Assign the uppercase letters of the alphabet to your variable.
3. Using the above string manipulation functions, print the following;
    a. "RSTUV".
    b. "BITS".
    c. "I love school".
    d. The numeric position of the letter "U".
    e. The number of times the letter "X" occurs in your string.
        i. Add a few extra Xs to your string to test fully.
    f. "rstuv" (`.lower()` function)

PS: You can also write secret messages with Python string manipulation functions. Download sm.txt to see...

This concludes chapter 9 of BITS Tutorial.