

ECDL COMPUTING

Syllabus 1.0
Learning Material



Provided by:
ECDL Malta



Copyright ECDL Foundation 2015 - 2019. All rights reserved. Reproducing, repurposing, or distributing this courseware without the permission of ECDL Foundation is prohibited.

ECDL Foundation, ECDL Europe, ECDL and related logos are registered business names and/or trademarks of ECDL Foundation.

This courseware may be used to assist candidates to prepare for the ECDL Foundation Certification Programme as titled on the courseware. ECDL Foundation does not warrant that the use of this courseware publication will ensure passing of the tests for that ECDL Foundation Certification Programme.

The material contained in this courseware does not guarantee that candidates will pass the test for the ECDL Foundation Certification Programme. Any and all assessment items and / or performance-based exercises contained in this courseware relate solely to this publication and do not constitute or imply certification by ECDL Foundation in respect of the ECDL Foundation Certification Programme or any other ECDL Foundation test. This material does not constitute certification and does not lead to certification through any other process than official ECDL Foundation certification testing.

Candidates using this courseware must be registered with the National Operator before undertaking a test for an ECDL Foundation Certification Programme. Without a valid registration, the test(s) cannot be undertaken and no certificate, nor any other form of recognition, can be given to a candidate. Registration should be undertaken at an Approved Test Centre.

Python is a registered trademark of the Python Software Foundation. Python and its standard libraries are distributed under the Python License. Details are correct as of December 2016. Online tools and resources are subject to frequent update and change.

ECDL Computing

With the increased use of computers in all areas of life, there is a growing interest in learning about the fundamentals of computing, including the ability to use computational thinking and coding to create computer programs.

The ECDL Computing module sets out the skills and competences relating to computational thinking and coding and will guide you through the process of problem solving and creating simple computer programs. Based on the ECDL Computing syllabus, this module will help you understand how to use computational thinking techniques to identify, analyse and solve problems, as well as how to design, write and test simple computer programs using well structured, efficient and accurate code.

On completion of this module you will be able to:

- Understand key concepts relating to computing and the typical activities involved in creating a program.
- Understand and use computational thinking techniques like problem decomposition, pattern recognition, abstraction and algorithms to analyse a problem and develop solutions.
- Write, test and modify algorithms for a program using flowcharts and pseudocode.
- Understand key principles and terms associated with coding and the importance of well-structured and documented code.
- Understand and use programming constructs like variables, data types, and logic in a program.
- Improve efficiency and functionality by using iteration, conditional statements, procedures and functions, as well as events and commands in a program.
- Test and debug a program and ensure it meets requirements before release.

What are the benefits of this module?

ECDL Computing has been developed with input from computer users, subject matter experts, and practising computing professionals from all over the world to ensure the relevance and range of module content. It is useful for anyone interested in developing generic problem solving skills and it also provides fundamental concepts and skills needed by anyone interested in developing specialised IT skills.

Once you have developed the skills and knowledge set out in this book, you will be in a position to become certified in an international standard in this area – ECDL Computing.

For details of the specific areas of the ECDL Computing syllabus covered in each section of this book, refer to the ECDL Computing syllabus map at the end of the learning materials book.

ECDL COMPUTING

LESSON 1 – THINKING LIKE A PROGRAMMER	1
1.1 Computational Thinking	2
1.2 Instructing a Computer	9
1.3 Review Exercise	12
LESSON 2 – SOFTWARE DEVELOPMENT	14
2.1 Precision of Language	15
2.2 Computer Languages	16
2.3 Text About Code	18
2.4 Stages in Developing a Program	20
2.5 Review Exercise	22
LESSON 3 – ALGORITHMS	23
3.1 Steps in an Algorithm	24
3.2 Methods to Represent a Problem	25
3.3 Flowcharts	27
3.4 Pseudocode	31
3.5 Fixing Algorithms	32
3.6 Review Exercise	34
LESSON 4 - GETTING STARTED	36
4.1 Introducing Python	37
4.2 Exploring Python	37
4.3 Saving a Program	40
4.4 Review Exercise	45
LESSON 5 - PERFORMING CALCULATIONS	46
5.1 Performing Calculations with Python	47
5.2 Precedence of Operators	48
5.3 Review Exercise	50
LESSON 6 – DATA TYPES AND VARIABLES	51
6.1 Data Types	52
6.2 Variables	53
6.3 Beyond Numbers	56
6.4 Review Exercise	60
LESSON 7 – TRUE OR FALSE	62

7.1 Boolean Expressions	63
7.2 Comparison Operators.....	64
7.3 Boolean Operators.....	65
7.4 Booleans and Variables	68
7.5 Putting It All Together	70
7.6 Review Exercise	72
LESSON 8 – AGGREGATE DATA TYPES.....	73
8.1 Aggregate Data Types in Python	74
8.2 Lists	75
8.3 Tuples.....	76
8.4 Review Exercise	78
LESSON 9 – ENHANCE YOUR CODE	79
9.1 Readable Code.....	80
9.2 Comments	80
9.3 Organisation of Code	81
9.4 Descriptive Names.....	81
9.5 Review Exercise	84
LESSON 10 – CONDITIONAL STATEMENTS.....	85
10.1 Sequence and Statements.....	86
10.2 IF Statement	86
10.3 IF...ELSE Statement.....	88
10.4 Review Exercise	90
LESSON 11 – PROCEDURES AND FUNCTIONS.....	91
11.1 Subroutines.....	92
11.2 Functions and Procedures	92
11.3 Review Exercise	96
LESSON 12 – LOOPS.....	97
12.1 Looping.....	98
12.2 Looping with Variables	100
12.3 Variations on Loops	101
12.4 Putting It All Together.....	103
12.5 Review Exercise	107
LESSON 13 – LIBRARIES	108
13.1 Using Libraries	109
13.2 Standard Libraries.....	110

13.3 Events.....	115
13.4 Pygame Library.....	117
13.5 Boilerplate Code	119
13.6 Drawing Using the Libraries	123
13.7 Review Exercise	128
LESSON 14 – RECURSION.....	129
14.1 Recursion.....	130
14.2 Recursive Drawing.....	132
14.3 Review Exercise	136
LESSON 15 – TESTING AND MODIFICATION	137
15.1 Types of Errors	138
15.2 Finding Errors	139
15.3 Testing and Debugging a Program.....	143
15.4 Improving a Program.....	145
15.5 Review Exercise	148
ECDL SYLLABUS.....	150

LESSON 1 – THINKING LIKE A PROGRAMMER

After completing this lesson, you should be able to:

- Define the term computing
- Define the term computational thinking
- Outline the typical methods used in computational thinking: decomposition, pattern recognition, abstraction, algorithms
- Use problem decomposition to break down data, processes, or a complex problem into smaller parts
- Define the term program
- Understand how algorithms are used in computational thinking
- Identify patterns among small, decomposed problems
- Use abstraction to filter out unnecessary details when analysing a problem

1.1 COMPUTATIONAL THINKING

Concepts

Computing is the performing of calculations or processing of data, especially using a computer system.

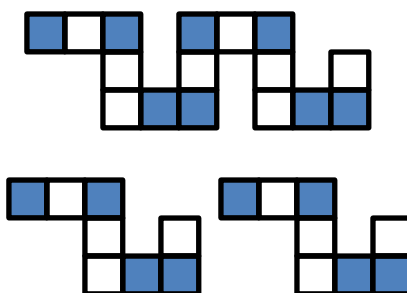
Computational thinking is the process of analysing problems and challenges and identifying possible solutions to help solve them.

Computational thinking is useful in complex problems or tasks such as designing products, cooking, planning events, fixing things that are broken, assembling flat pack furniture, and in many other problem solving situations.

Computational thinking uses four key problem-solving techniques. They can be used in any order and in any combination.

Pattern Recognition

A pattern is a repeated design or feature, like the chorus of a song or a motif on fabric or wallpaper. Patterns can also be found in activities, for example, several different recipes may involve setting the oven to a certain temperature and waiting for it to heat up. This is known as a shared pattern. Pattern recognition involves finding patterns or repetition within complex problems or among smaller related problems.



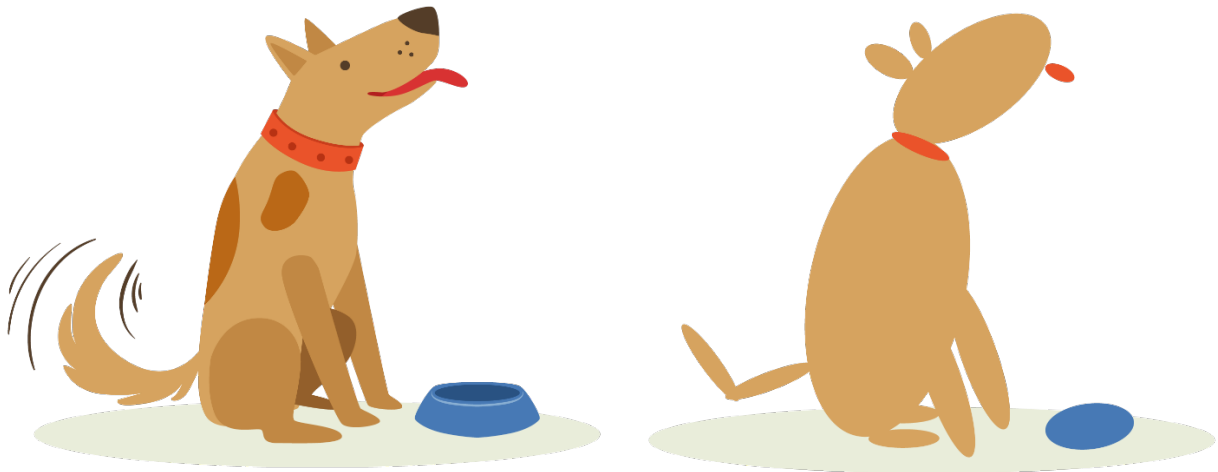
In the figure above the first set of coloured blocks has been split in two to highlight the presence of a pattern that is being repeated.

Abstraction

Abstraction is the process of extracting the most important or defining features from a problem or challenge. The extracted features provide the information to begin examining the challenge and find potential solutions. It involves filtering out unnecessary details and only looking at information that is relevant to solving the problem. In the task of baking biscuits, it is not important whether you are right or left-handed. The information that is relevant to completing the task includes the

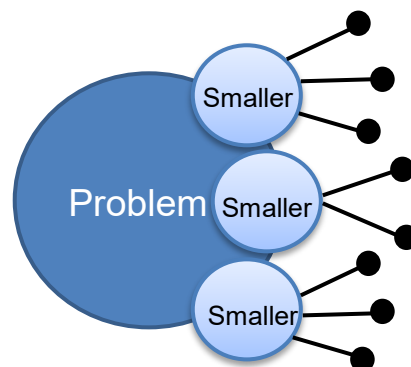
ingredients, the order in which you mix them in, and the duration and temperature of cooking.

In the following example, any unnecessary details have been removed from the second image of the dog so only the information relevant to solving the problem remains – there is a dog and it wants food.



Decomposition

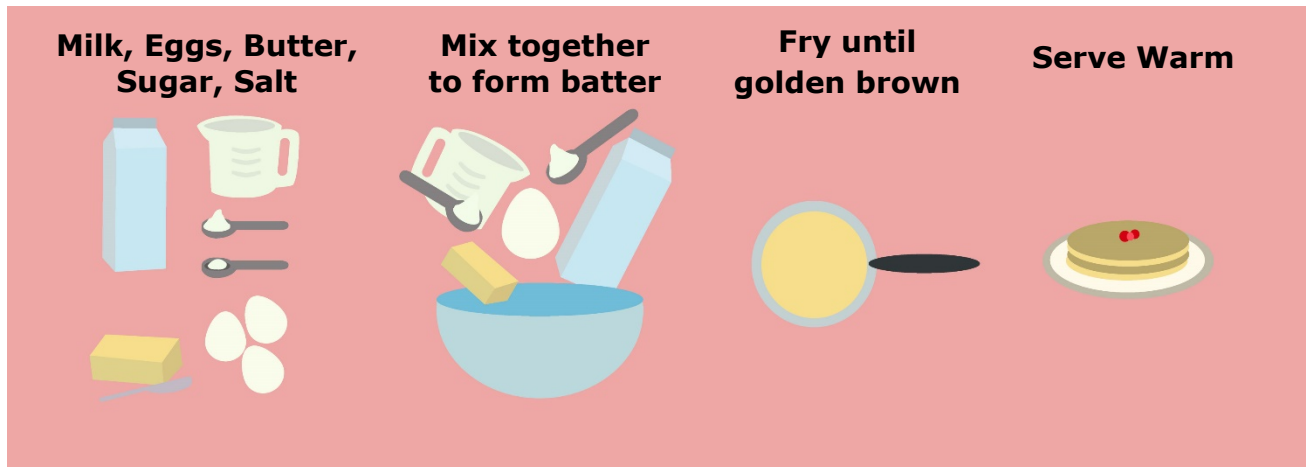
Decomposition involves breaking a complex problem down into smaller, simpler, easier to understand problems. The smaller problems can be broken down into smaller and smaller problems until they are easy to understand and manage. In the task of making biscuits, a smaller problem might be to ensure that the oven is at the right temperature. The image illustrates how a complex problem can be decomposed into smaller and smaller parts or problems.



Algorithms

An algorithm is an organised set of instructions which provides steps for solving a problem or completing a task. For example, an algorithm could be the instructions in a recipe or the calculations for a computer to follow. Algorithmic design is the process of creating these well-defined instructions in the form of steps to solve

problems or complete tasks successfully. A possible algorithm for making pancakes involves the following steps:



Two additional approaches often included as part of computational thinking are evaluation and generalisation:

Evaluation

Evaluation involves validating the design, of a product or an algorithm, by checking that it does what it's meant to, or solves the problem.



Generalisation

Generalisation is taking a solution and finding a way to make it useful in a wider set of circumstances. For example, you might use symbols instead of words on a products' controls so it can be used regardless of the language spoken. In the following image the basic structure of the flower is always the same and can be used repeatedly but the characteristics such as the colour and shape can be varied.



These six computational thinking skills work together to solve complex problems in computing and beyond.



Steps

Example: Designing a Washing Machine

In this example the techniques of computational thinking are applied to the complex problem of designing a washing machine.



Abstraction

The first step in designing a washing machine is to determine the goal – for example it should be able to wash vigorously or gently, at a low or high temperature and produce clean clothes. The design phase involves listing the features to achieve this goal. Abstraction can be used to help the designer filter out what is relevant and what is not when listing the features to include and exclude. Details that aren't relevant might include things such as the colour of the washing machine or whether the load to be washed has six pairs of socks or three pairs of socks.

Details that are relevant are those that affect the overall functionality and they might include things such as water temperature and whether a programme is for delicate or hard-wearing fabrics. These details are relevant to solving the problem of producing clean clothes without damaging them.

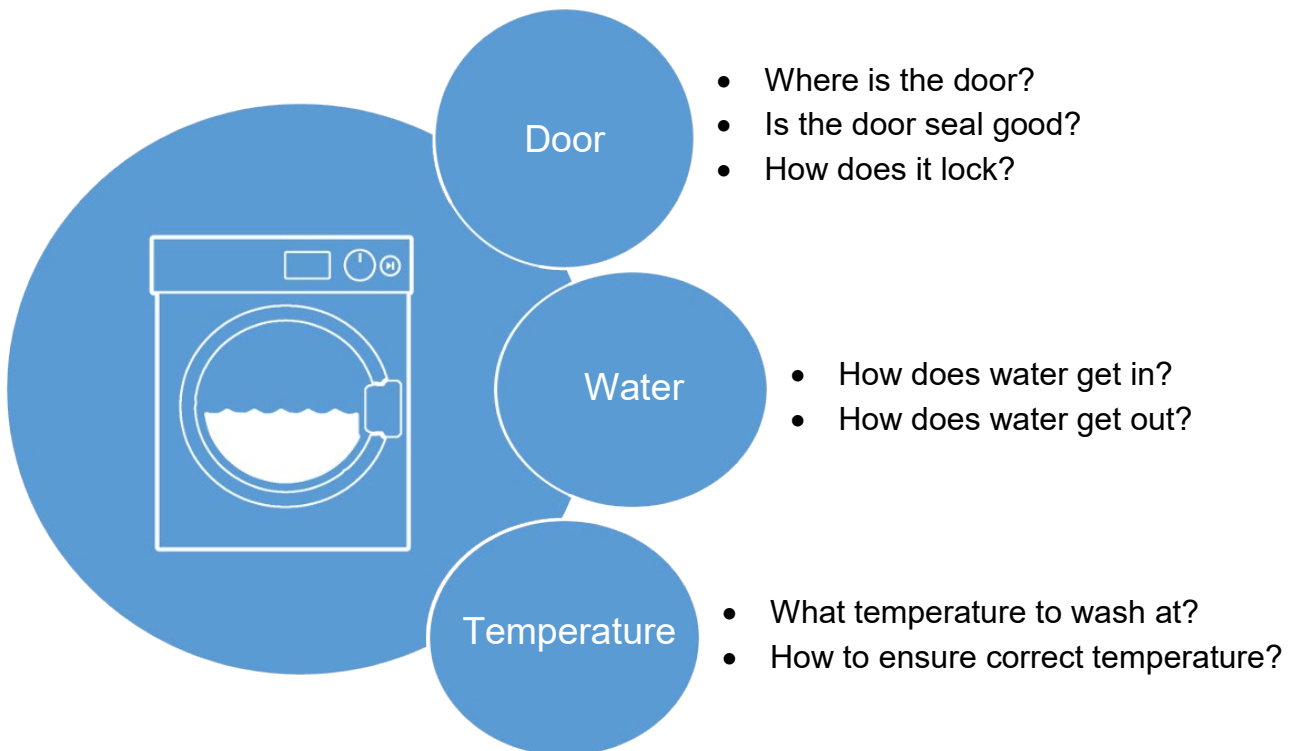
Decomposition

Decomposition can be used to break the problem down into smaller, more manageable problems, such as:

- How do we get clothes in and out of the washing machine?
- How do we get water in and out of the washing machine?
- How do we make sure the water is at the right temperature?

In turn these small problems may be decomposed into even smaller ones:

- Where is a good place for a door on a washing machine?
- How can we make a door that seals properly, so water does not leak out?



Decomposition of the problem of designing a washing machine

Algorithms

Algorithms can be designed to specify the precise steps that the machine should follow for different wash cycles. And there are many other algorithms needed. For example an algorithm can be designed to specify the sequence of precise steps to be followed to manufacture the washing machine.

Evaluation

Designing a product is a big challenge, so early designs will be continuously evaluated. Through evaluation, the designers learn where and how the design can

be improved. For example, if while testing a machine, water leaks out, the design can be modified to prevent this happening in future.

Generalisation

The washing machine design may also be improved by making it universal, for example using symbols instead of words on the controls, so that it can be used by people regardless of their language, or by designing it to work on 110 volts as well as 220 volts for use in different countries. If the design is modified in this way, then it has been generalised.

Pattern Recognition

Pattern recognition has an important role and along with the other five techniques it can be used multiple times throughout the different stages of the design process. It can help with abstraction by highlighting the similarities and differences in aspects of a problem. Pattern recognition can also help with generalisation; where aspects of one particular problem can be related to more general problems. Similarly, evaluation of the design helps to generalise it, by evaluating how the design may be applied in different settings.

The six techniques in computational thinking are not necessarily applied in sequence or at only one stage of the design process. They are used in many stages of design and in different sequences.



Steps

Example: Organising a Music Festival

In this example the techniques of computational thinking are applied to the complex problem of organising a music festival.

Abstraction

To begin to plan a music festival you must understand what a music festival is, so the first step could be to list essential things that a music festival should have:

- Musicians
- Venue
- Marketing and publicity
- Tickets
- Staff etc.

There will be other details to consider in organising the festival, which are not essential such as the colour of the tickets and the exact wording on the tickets.

These can get in the way of the initial planning, and can be worked out later. Using **abstraction**, the non-essential details can be removed from the plan for the moment.

Decomposition and Algorithms

The challenges in organising the musical festival can then be broken down into smaller challenges. In practice, they could be delegated to individual people to solve:

- Someone to liaise with the venue where the festival will take place.
- Someone to deal with publicity.
- Someone to work on bookings and tickets.

These smaller challenges can be further broken down into even smaller problems to solve. This is **decomposition** of the problem and an example of applying the methods of computational thinking.

For example the problem of organising the venue could be broken down as follows:

- Where will it be?
- When is it available?
- What are the car parking arrangements?

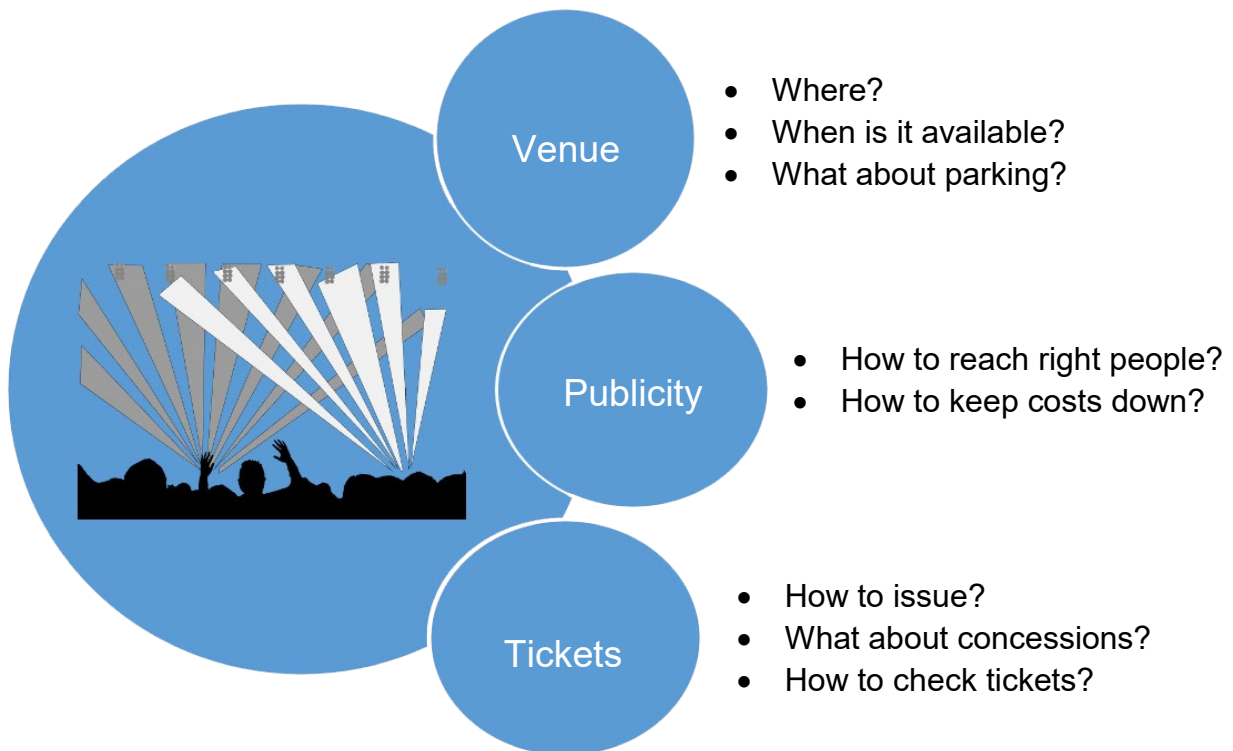
And the smaller problem of the car parking arrangements can be broken down into small processes or **decomposed**. And simple **algorithms** can be created, including some of the following:

- To direct cars to designated spaces.
- To direct cars to the overflow car park when the main one is full.
- To prevent cars blocking important routes.
- To make sure priority parking is kept clear for people who are entitled to it.

The problem of publicity could be broken down as follows:

- How to make the festival interesting so that people will want to go?
- How to advertise to your intended audience event in a cost-effective way?

And the booking and ticketing process can also be **decomposed** further. For example the ticketing process can be broken down into a set of steps, and an **algorithm** created covering design, printing, purchase, delivery, concessions, refunds etc.



Decomposition of the problem of organising a music festival

Evaluation

After the event has run, it can be helpful to gather feedback about what worked well and what did not work so well. This is a way of evaluating the plan. If the festival runs again, the lessons learned from the evaluation will improve the following year's event.

Generalisation

Generalisation is relevant too. A team that has run a music festival might go on to more ambitious goals, for example a series of festivals in a tour, or increasing the duration of the festival, adding more acts, or more kinds of music. The existing solution or formula is reused, but generalised to include new elements.

1.2 INSTRUCTING A COMPUTER

Concepts

Computational thinking is a general problem solving approach but it can also be used as the starting point in creating instructions for computers.

The decomposition of problems into simpler steps leads to development of one or more **algorithms** - collections of well defined, simple steps, to be followed in order to solve a problem. Algorithms can then be presented in ways that computers can understand.



Steps

Example: An algorithm for sorting people by height



In this example you want to sort the people in your class by height. To do this, you could decide on a number of steps to perform:

- Step 1: Line everyone up in a single row.
- Step 2: Decide which end of the row is to be 'tall' and which is to be 'short'.
- Step 3: Repeatedly compare heights and switch students when they are in the wrong place.

This set of steps is an algorithm. However, it is not very detailed. Step 3, for example, could be further decomposed into smaller problems:

- Which end of the row to start at
- How to compare student heights
- What to do if the students are taller or shorter

Once designed correctly, this algorithm for sorting people by height could be modified to solve slightly different problems. A small variation on the algorithm could instead sort people in the class by order of birthday. And with further modifications, it could be used to find out if any two people have the same birthday.



Concepts

Algorithms for Computers

In the algorithm for sorting people by height the instructions are detailed enough for a human to follow, but they aren't detailed enough for a computer to follow. A computer needs to be instructed using a computer language, which is much more detailed and precise than human language.

So for a computer to follow the instructions in the algorithm, the algorithm must be converted into a language that the computer can understand. An algorithm expressed in a form that can be understood and executed by a computer is known as a **program**.

When the computer obeys the steps in a program, it is said to run or execute the program. So a computer program is written in a computer language and the computer works by running or executing the program.

Imagine a robot baking a cake. An instruction in an algorithm to 'put the cake in the oven' would not be detailed enough for the robot (computer) to act on. A robot chef requires a program with much more detailed instructions about how to move its arms and fingers when putting the cake in the oven - and how not to topple over in the process.

1.3 REVIEW EXERCISE

1. Computing is a set of activities that include:
 - a. performing of calculations or processing of data.
 - b. following a recipe to make biscuits
 - c. accurate and careful observation of the stars, over a long period of time.
 - d. accurate and careful observation of chemical reactions.

2. Computational thinking is:
 - a. When computers are solving a difficult problem.
 - b. the process of analysing problems and challenges and identifying possible solutions to help solve them.
 - c. Using a computer to do lots of maths.
 - d. Any activity where a computer and a person work together to do more than either could on their own.

3. Which method is not a typical part of computational thinking?
 - a. Abstraction
 - b. Apprehension
 - c. Decomposition
 - d. Pattern Recognition

4. Which of these is a good example of decomposition of a problem?
 - a. Splitting the task of designing a robot into designing the hand, designing the power source, deciding on and designing the sensors.
 - b. Slicing a cake into 6 equal slices
 - c. Putting an apple into a glass jar and taking photos of it each day to see what happens.
 - d. Using a search engine to find the answer to a question.

5. A program is:
 - a. The detailed rules and regulations of a game or sport
 - b. An algorithm expressed in a form that is suitable for a computer
 - c. The collection of laws and regulations that determine what buildings can be built legally at a particular place.
 - d. A sequence of instructions for a person to follow, e.g. a recipe for baking a cake.

6. Which of these is least likely to be a way that algorithms are used in computational thinking?
 - a. An algorithm may lead to a computer program that when run solves the problem
 - b. An algorithm may lead to a computer program that employs intuition to derive better solutions.
 - c. An algorithm may provide a step by step guide for manufacturing something.
 - d. An algorithm may provide a step by step guide for processes involving people, money and resources.

7. Here are three recipes:

Chocolate cake:

Set oven to 180°C
Butter two 9" cake pans
Mix ingredients with a whisk for one minute
Transfer mix to cake pans
Bake for 30 mins
Allow to cool
Ice and decorate cake.

Gingerbread men:

Cream the ingredients together
Set oven to 190°C
Roll out the dough to about 1/8" thickness, and cut into gingerbread men shapes.
Bake until edges are firm, about 10 minutes.

Blueberry muffins:

Set oven to 185°C
Grease 18 muffin cups
Cream butter and sugar until fluffy
Add other ingredients
Spoon into muffin cups
Sprinkle with topping
Bake for 15 to 20 mins.

Which of these is a pattern common to all three recipes?

- a. Grease 18 muffin cups.
 - b. Ice and decorate.
 - c. Roll out the dough.
 - d. Set oven to some temperature.
8. For designing a program for cooking for a robot chef to use, which of the following would be the most relevant detail?
- a. The colour of the robot.
 - b. What quantities of each ingredient to use.
 - c. Which shop the ingredients were bought from.
 - d. Whether the programmer is right or left handed.

LESSON 2 – SOFTWARE DEVELOPMENT

After completing this lesson, you should be able to:

- Understand the difference between a formal language and a natural language
- Define the term code; distinguish between source code and machine code
- Understand the terms program description, specification
- Recognise typical activities in the creation of a program: analysis, design, programming, testing, enhancement

2.1 PRECISION OF LANGUAGE



Concepts

Types of Language:

Natural Language

Spoken languages like English, French or Chinese are natural languages. A natural language needs context to be clearly understood and avoid ambiguity.

Formal Language

A formal language is strictly structured, with exact and precise rules. It is used in mathematics, chemical equations and computer programs. It is clear and unambiguous.

Computer languages are formal languages. One way formal languages can avoid ambiguity is by using brackets to group words and terms, and by avoiding words like 'it' or 'he' or 'she' where it might not be clear what 'it', 'he' or 'she' refers to.

Here are three different types of brackets used in formal languages. Each type of brackets has a different function, and they can be used to define things like which order a set of steps are expected to be performed in a program.

BRACKET	NAME
()	Round brackets, or parentheses .
{ }	Curly brackets or braces .
[]	Square brackets.

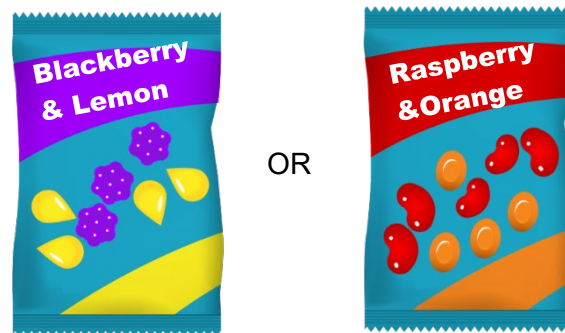


Steps

Example: Ambiguity with 'and' and 'or'

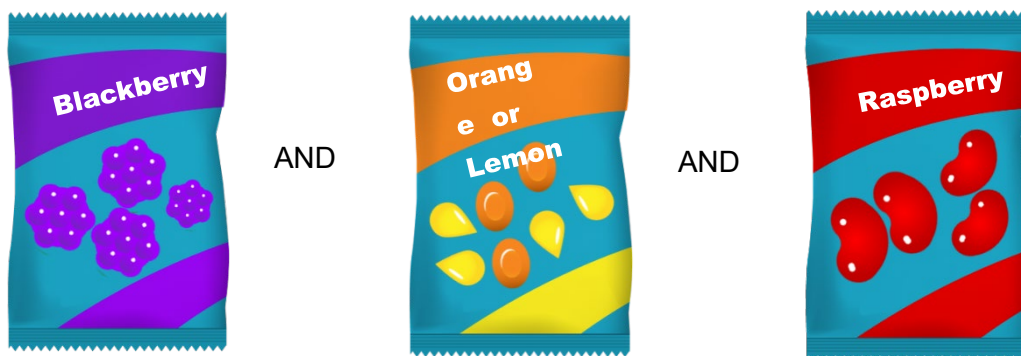
If you ask someone what flavour of sweets they would like, they might say: "Blackberry and Lemon or Raspberry and Orange". You understand the response, but for a computer to understand the sentence, the words need to be grouped correctly.

For the computer, you need to write (Blackberry AND Lemon) OR (Orange AND Raspberry)



(Blackberry AND Lemon) OR (Orange AND Raspberry)

Whereas if you write Blackberry AND (Orange OR Lemon) AND Raspberry, the computer interprets it as:



Blackberry AND (Orange OR Lemon) AND Raspberry

2.2 COMPUTER LANGUAGES



Concepts

Computer languages are designed for writing computer programs that instruct computers to follow a sequence of steps to solve a problem. Computer languages have a smaller vocabulary than natural or spoken languages.

There are many different computer languages. These learning materials use a popular computer language called Python. Some other common languages are Java and C++.



Concepts

Computer Code

The text of a computer program, written as a series of instructions for a computer to execute, is known as code. Different computer languages use different styles of code with different rules and ways of organising instructions, often referred to as syntax.

The work of writing a program is called programming or coding.

People who write programs are called programmers or coders.

There are two kinds of computer code: **source code** and **machine code**.

Source code is the code written by the programmer that humans can understand. It is typed into a computer, usually as text, punctuation and symbols, which contains instructions for the computer. If you learn the formal programming language and its rules (e.g. Python), you can write source code.

Machine code is a series of 1's and 0's created by the computer from the source code that the computers electronic circuits can understand. Creating machine code from source code is known as compiling or interpreting the source code.



Steps

Example: Machine code to lock a door

The instruction to lock a door written in source code might look like this:

Lock(door)

Although this isn't grammatically correct, it does make some sense to people reading it.

A computer reading the source code instructions sees the letters 'L', 'o', 'c', 'k', '(', and so on. The instructions aren't in a format the computer can act on directly. Instead the computer needs the instructions to be converted into a format that it can understand and obey.

The computer has electronic circuits that act on certain patterns or combinations of 1's and 0's. Below is an example of what some patterns of 1's and 0's might mean.

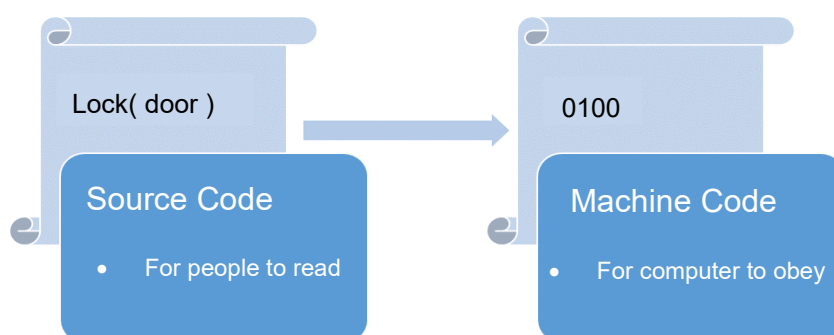
Machine Code	Meaning of the machine code instruction
0001	Sound alarm
0010	Lock windows
0100	Lock door
0110	Lock windows and door

1000	Turn on sprinkler system
1001	Turn on sprinkler system and sound the alarm

Instructions like '0110' are machine code instructions.

In reality the full range of possible machine code instructions for a computer would be much larger. There would, for example, be many instructions for performing arithmetic.

Source code is converted into machine code before a computer obeys the instructions. 'Lock(door)' would be translated to 0100 and the computer could then obey that instruction.



Source Code v Machine Code

In this simple example, one source code instruction 'Lock(door)' corresponds to one machine code instruction, 0100. Usually, a single line of source code requires several machine code instructions one after the other, in the correct sequence.

The earliest computer programmers converted the source code into machine code manually, and created and documented the series of 1's and 0's themselves. The invention of programs to translate source code to machine code made it possible to write larger and more complex computer programs.

2.3 TEXT ABOUT CODE

Concepts

As well as writing source code programmers also need to document their work in the form of text notes. These text notes are not read by the computer, but help the programmer and other colleagues to understand the different stages of the program.

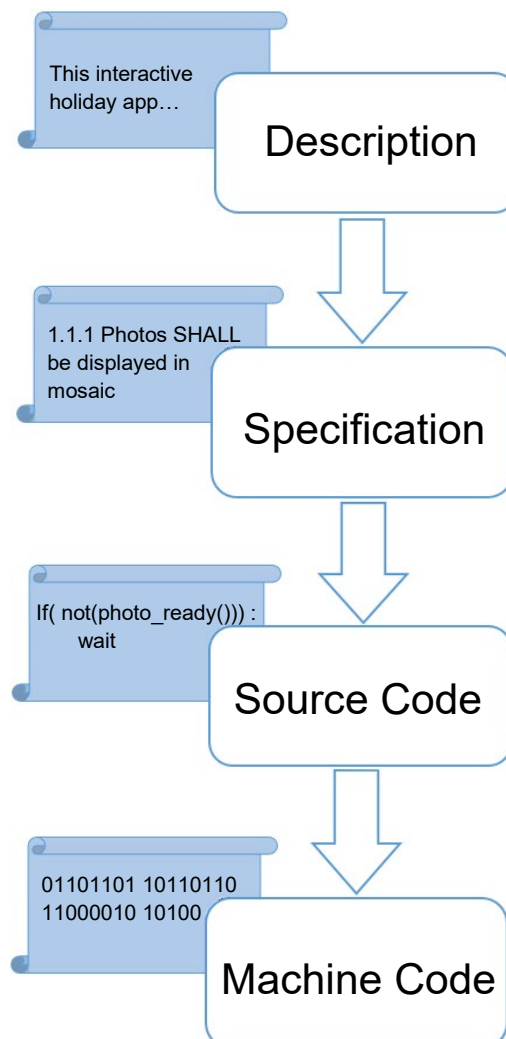
Text Describing a Program

Programmers write program descriptions and specifications to help everyone involved in a project to understand what the program was designed to do and what problem needed to be solved. They can also be used in the evaluation stage to check that the program is working as desired.

A **program description** explains what a program is designed to do and how it works. This could be helpful for other programmers, the user of the end product and also the marketing department to use for promotion and sales.

A **program specification** is the set of requirements that outline what a program will do. Usually, the specification is written before the program. A specification is more detailed than a description, and it states requirements that can be tested.

For example, a specification might include the requirement “The program SHALL dim the screen if the computer has not been used for one minute, to save electricity”. Once the program is written, the program can be tested by leaving the computer running for a minute and checking that the program does indeed dim the screen. The following image describes the possible progression in work on a computer program from development to execution.

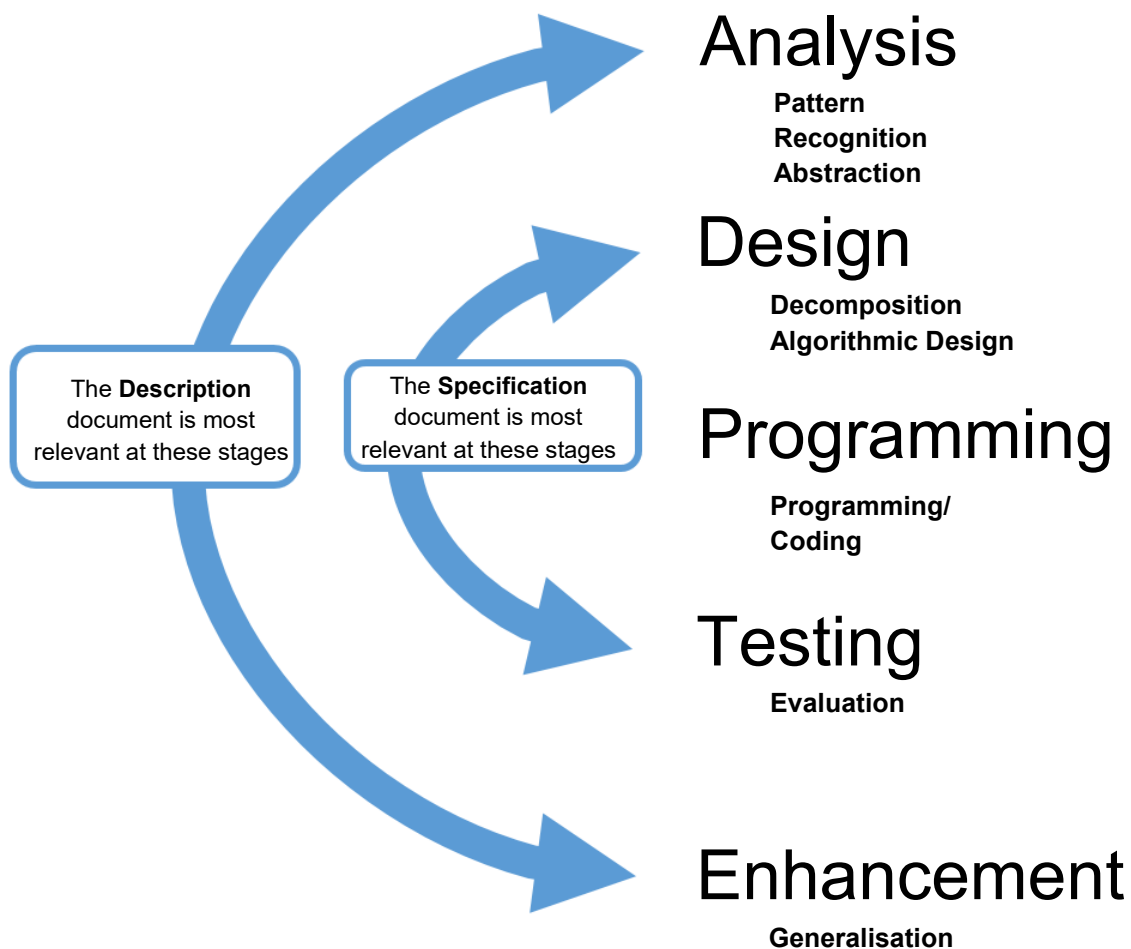


2.4 STAGES IN DEVELOPING A PROGRAM

Concepts

Descriptions and specifications can play an important role in the development of a program.

The following diagram shows some of the main stages and activities in the creation of a program.



Stages in Program Development

Analysis

This involves defining the problems to be solved more clearly. Analysis is largely a process of abstraction, listing all aspects of the problem, identifying the relevant aspects and how they relate to each other.

Design

This involves working out the algorithms (sets of steps to follow) to solve the problem. This uses decomposition to break the problem down into smaller parts, and planning the design of the algorithm.

Programming

This involves writing the program. This includes finding a way to express the algorithm in the chosen computer language.

Testing

This involves checking that the program actually does what it was intended to. Testing can uncover errors in the logic or syntax of the program. These are covered in more detail in Lesson 15.

Enhancement

This involves adding new features to extend the program's capabilities, improve its performance or functionality, or to generalise it for use in different situations.

A general **program description** is typically formulated very early in the project, as part of deciding the scope of what the program will do.

The **program specification** is normally drawn up during the design phase. Some of the requirements in the specification may arise directly from decomposition of the problem into smaller problems. The program specification is examined during testing, as each statement about the program's capabilities must be verified.

In planning enhancements, the program description can be updated to record plans for extending the program's capabilities.

2.5 REVIEW EXERCISE

1. A formal language is:
 - a. A language in which it is not possible to make a mistake.
 - b. A language with defined rules and unambiguous meanings.
 - c. The best language to use when writing a postcard.
 - d. Any language that has the words "and" and "or" in it.

2. English, Arabic and Chinese are:
 - a. Formal languages.
 - b. Computer languages.
 - c. Natural languages.
 - d. Source code.

3. Machine code is.
 - a. Translated into source code so that a computer can obey the instructions.
 - b. A method for secure communication between two computers.
 - c. The 1's and 0's that a computer obeys.
 - d. A written agreement between a programmer and a customer specifying what a program should do.

4. An algorithm written in the python computer language is an example of:
 - a. Source code
 - b. Machine code
 - c. A program specification
 - d. A program description

5. A program specification is:
 - a. The code that the computer runs.
 - b. The comments in code that a programmer reads.
 - c. The actual electrical signals in a computer, when a program is running.
 - d. A description of what a program should do that is used during designing the program.

6. Match each activity in creating a program, a to e, to their purpose:
 - a) Testing, b) Design, c) Programming, d) Analysis, e) Enhancement

Improving an existing program.	
Clearly defining the problem.	
Checking whether the program works correctly.	
Expressing the algorithm in the chosen computer language.	
Working out an algorithmic approach to solving a problem.	

LESSON 3 – ALGORITHMS

After completing this lesson, you should be able to:

- Define the programming construct term sequence. Outline the purpose of sequencing when designing algorithms
- Recognise some methods for problem representation like: flowcharts, pseudocode
- Recognise flowchart symbols like: start/stop, process, decision, input/output, connector, arrow
- Outline the sequence of operations represented by a flowchart, pseudocode
- Write an accurate algorithm based on a description using a technique like: flowchart, pseudocode
- Fix errors in an algorithm like: missing program element, incorrect sequence, incorrect decision outcome

3.1 STEPS IN AN ALGORITHM



Concepts

Sequencing

Computer algorithms are complex problems broken down into simpler steps or **instructions**. In most algorithms, the instructions are obeyed one after the other in a **sequence**.

A **sequence** is a number of simple instructions that should be carried out one after the other.

In most algorithms the order of the instructions matter. A robot chef would need to put the cake mixture into the tin before putting it in the oven.

Designing sequences of instructions is a crucial skill in programming. A programmer needs to make sure that all the required actions are performed in the right order to accomplish a task or set of tasks. A sequence is the fundamental method of control in a computer program. The sequence is decided upon in the design phase of a project where algorithms and flowcharts are used to create the most efficient and correct sequence of program control.

Input and Output

Instructions often use information from the world outside the computer and do something with it. Computer programs can receive information to work on via **inputs** and give results via **outputs**.

An input

A value from outside the computer that is needed for the instructions to be followed. For example: a measurement of temperature, or a number typed on the keypad of a security/alarm system.

An output

A value calculated, or an action performed, by the computer program, which is displayed to the world outside of it. For example: a light being switched on or off, a security alarm being set, or a message displayed on a screen.



Steps

Example: Inputs and Outputs of a Mobile Phone

A mobile phone takes in information or inputs and processes them and returns results or outputs.

- The touch screen is an important input method, for example when you use it to type a message or tap open an application.
- The screen display is an important output, for example when displaying a list of contacts.

Yes / No Decisions

As you have seen in previous lessons, the order of instructions, or sequence, is important in computing. In general, instructions are followed one after the other, but sometimes there are decisions to be made on what to do next. The algorithm can be designed to ask questions and expect an input to help decide on the next step.

A **yes / no decision** is an instruction that chooses the next instruction to obey. It does this based on whether the answer to a question is yes or no.



Steps

Example: Waiting for an Oven to Heat Up

Before putting a cake mixture into the oven, the oven needs to be at the right temperature. An algorithm could include a yes / no decision to determine this. “Is the oven at the right temperature?” A thermostat will provide the input, yes or no. If the answer is yes, then it is ok to put the cake into the oven. If the answer is no, the algorithm will wait a while and ask the question again.

3.2 METHODS TO REPRESENT A PROBLEM



Concepts

Two techniques to help programmers write programs are pseudocode and flowcharts. These are used to represent the sequence of steps in an algorithm. Algorithms expressed in pseudocode or as flowcharts are rarely expressed precisely enough for a computer to use, but present a series of steps in sufficient detail for a programmer to see how the algorithm is supposed to operate.

Pseudocode:

Pseudocode is an informal way of representing an algorithm, using written instructions in natural language, for example English. The steps in the algorithm are written out as a sequence of instructions. An algorithm written as pseudocode is intended for a human to read rather than a computer.

Pseudocode looks like the code that the program will be written in, but is not quite as precise. If the pseudocode makes sense to read, then the next step is to write the actual code for the program.

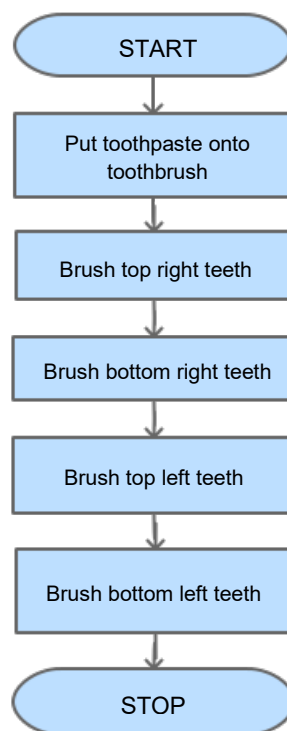
An algorithm for brushing teeth, written as pseudocode could be written as:

```
Put toothpaste onto toothbrush  
Brush top right teeth  
Brush bottom right teeth  
Brush top left teeth  
Brush bottom left teeth  
STOP
```

Flowchart

A flowchart is a pictorial way of representing an algorithm. Flowcharts represent a series of simple steps with arrows showing the progression from step to step. The steps are shown using shaped boxes containing text. A flowchart is a fundamental tool used in the development of computer programs. It allows developers to outline, step by step how they want to solve a problem or how they want a program to behave.

The algorithm for brushing teeth could also be shown as a flowchart:

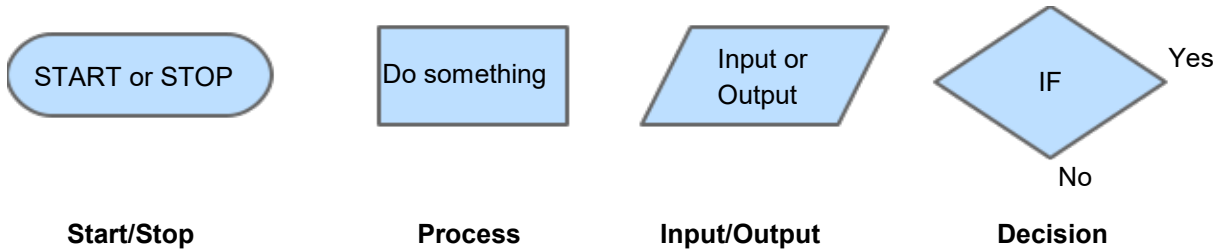


Flowchart for brushing teeth

3.3 FLOWCHARTS

Concepts

Some of the shapes used in a flowchart are shown below:



Flowchart symbols

The shapes represent the following:

Start or Stop

The algorithm starts at the Start box and runs until it reaches a Stop box.

Process

These boxes contain simple instructions to do something, such as add two numbers or look up a word in a dictionary.

Input or Output

These boxes are for interaction with the world outside the computer. For example, an input could be from a movement or temperature sensor. An output could be an action such as turning on or off a light.

Decision

Decision boxes allow for alternative choices in what the algorithm does next. They often have a question in them which will have a yes or no answer. If the answer is 'yes' one route is taken. If the answer is 'no' the other route is taken. This is how algorithms can go beyond simple sequences of steps.

Connecting the Boxes

These are two additional kinds of flowchart symbols:



The arrow and connector are used to connect the flowchart boxes.

Arrow

A directional line drawn to connect two boxes in a flowchart. This arrow shows the direction of flow in an algorithm. These are often referred to as flow lines. For example, instructions in a sequence have arrows between them.

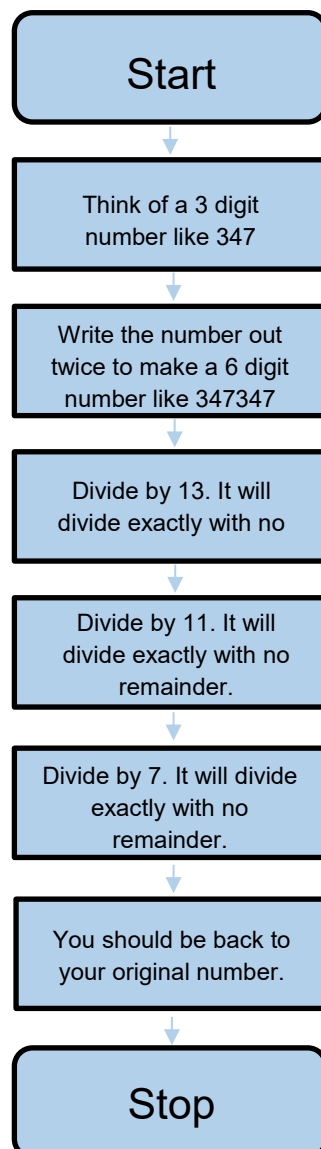
Connector

A small circle in a flowchart that is used to connect two flow lines together. It shows a jump from one point in the process flow to another, for example when answering a “Yes/No” question.



Steps

Example: Flowchart for the 347347 'Magic Trick'



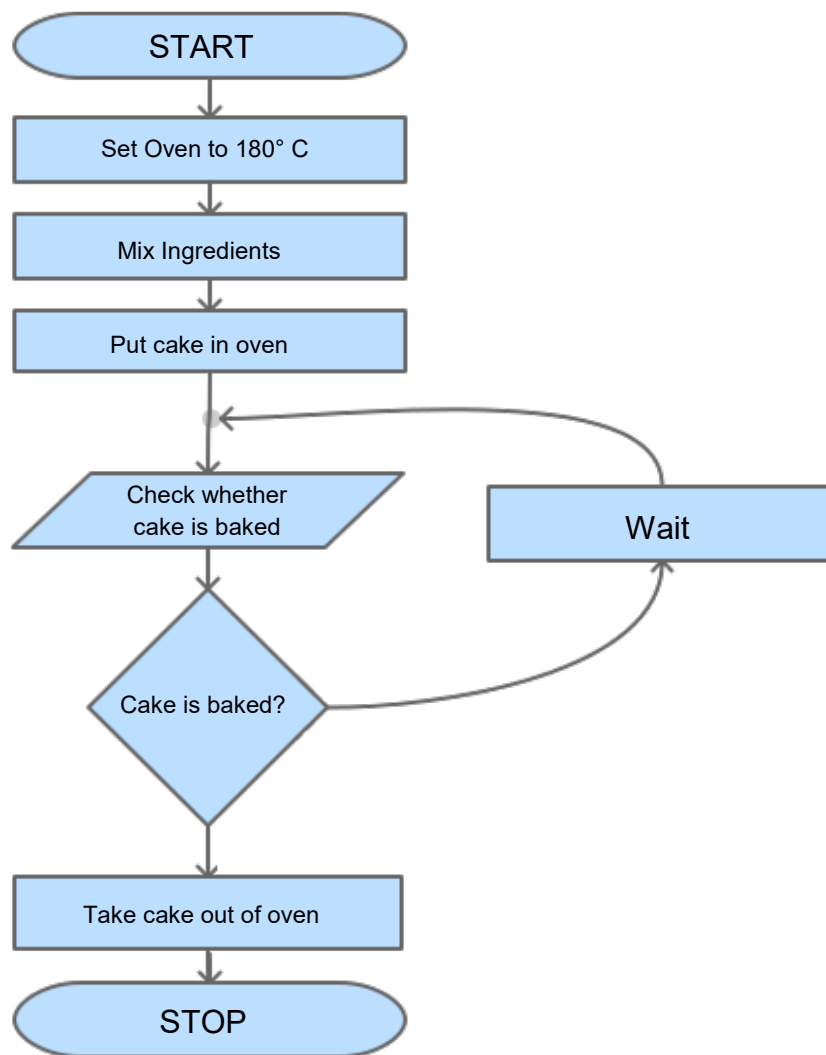
Flowchart for a numerical trick

This is a flowchart for a numerical magic trick. This flowchart shows a simple sequence, with no decision boxes. We begin the process from the Start box and continue following the flow lines completing the instructions in each of the boxes until we reach the Stop box.

Example: Flowchart for Baking a Cake

This flowchart represents an algorithm for baking a cake. It has a decision box in it. The decision box determines whether the cake is ready to take out of the oven, and if not instructs the chef to wait until it is.

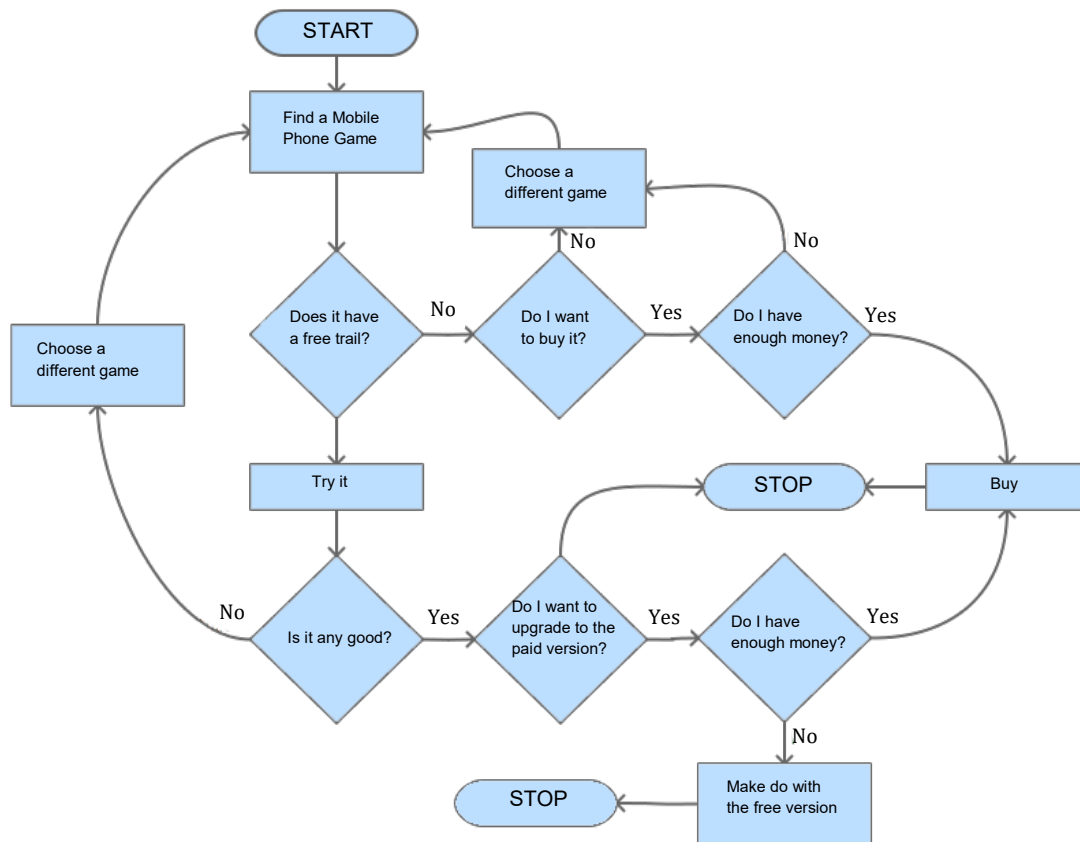
“Is cake baked?” is an input to the algorithm and has the input/output shaped box.



Flowchart for baking a cake

Example: Flowchart for choosing a Mobile Phone Game

This flowchart shows the sequence of actions for choosing a mobile phone game. Each of the diamond shaped boxes has a question in it. The answer to the question can be yes or no. The answer determines which box to go to next.



Flowchart for choosing a mobile phone game

Example: Flowchart for Getting up in the Morning

Make a flowchart for getting up in the morning, having breakfast, and getting to school.

HINTS:

1. Start with a sequence of actions and write those steps as pseudocode, in an ordered list.
2. Take some of the steps and decompose them into smaller steps detail. For example, if you have a step called 'make breakfast' - that can be broken down into smaller steps like "put bread in toaster".
3. Represent this more detailed sequence as a flowchart.

4. Make the flowchart more interesting, by introducing variation. For example, how does the weather affect getting to school? The variations may depend on questions like:
 - Is it raining?
 - Am I late for school?
5. Add decision boxes for these questions, with different steps if the answer is yes or no.

3.4 PSEUDOCODE



Concepts

Instead of creating a flowchart, an algorithm can be represented in pseudocode.

Pseudocode is a way to informally describe how an algorithm operates. It combines informal natural language (e.g. English) with some of the structure of programming languages.



Steps

Example: Pseudocode for the 347347 'Magic Trick'

The numerical magic trick flowchart could be written as pseudocode as follows:

```
Think of a 3 digit number like 347
Write the number out twice to make a 6 digit number like 347347
Divide by 13
Divide by 11
Divide by 7
You should be back to your original number.
STOP
```

Example: Pseudocode for Baking a Cake

Pseudocode for baking a cake is shown below:

```
Set oven to 180°C
Mix ingredients
Put cake in oven
Check whether cake is baked and if it isn't
    Wait
    Take cake out of oven
STOP
```

Pseudocode is written using the structure and some conventions of programming languages. The word 'Wait' is **indented**, which indicates that it only happens while the cake isn't baked.

3.5 FIXING ALGORITHMS



Concepts

When writing algorithms it is easy to make mistakes such as leaving out steps, putting steps in the wrong order, or making incorrect decisions. These errors must be corrected.

Here are some common errors and how to fix them.

1. Incorrect Sequence

Writing instructions in the wrong order is known as an incorrect sequence.

Here is an algorithm for brushing teeth, written in pseudocode. The instructions are in the wrong order. The algorithm can be fixed by moving 'Put toothpaste onto toothbrush' to the top.

```
Brush Top Right Teeth
Brush Bottom Right Teeth
Brush Top Left Teeth
Brush Bottom Left Teeth
Put toothpaste onto toothbrush
STOP
```

2. Incorrect Decision Outcome

Here is an algorithm for baking a cake and taking it out of the oven when it is baked and not before.

```
Set oven to 180°C
Mix ingredients
Put cake in oven
Is cake baked?
    Wait
Take cake out of oven
STOP
```


The decision to wait could be wrong because there is no test to see if the cake is actually baked. To correct a possible incorrect decision outcome, an extra line of pseudocode is needed:

```
Set oven to 180°C
Mix ingredients
Put cake in oven
Test with Fork
Is cake baked?
    Wait
Take cake out of oven
STOP
```

3. Missing Program Element

Here is an algorithm for baking a cake which is missing some important steps.

```
Set oven to 180°C.
Wait for oven to heat up.
Take cake out of the oven.
STOP
```

The algorithm can be fixed by adding the missing steps:

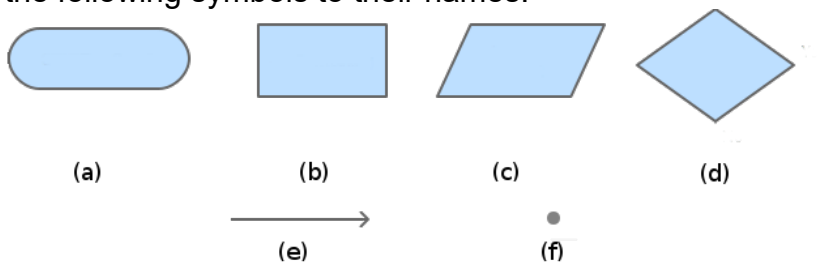
```
Set oven to 180°C.
Mix ingredients
Put the mixture into the cake tin
Wait for oven to heat up.
Put cake tin into oven
Wait for cake to cook.
Take cake out of the oven.
STOP
```

3.6 REVIEW EXERCISE

1. A sequence is:
 - a. A number of instructions that can be obeyed in any order.
 - b. A number of instructions that should be obeyed one after the other.
 - c. An analysis of a problem to be solved.
 - d. A collection of recommendations for enhancing a computer program.

2. Which of these is not used to represent an algorithm?
 - a. A computer program.
 - b. Flowchart.
 - c. Pseudocode.
 - d. Pseudoscience.

3. Match the following symbols to their names:



Arrow	
Decision	
Start or Stop	
Process	
Input or Output	
Connector	

4. In the following pseudocode which instruction is obeyed immediately after 'Mix Ingredients'?

```

Set oven to 180°C
Mix ingredients
Put cake in oven
Check whether cake is baked and while it isn't
    Wait
Take cake out of oven
STOP
    
```

- a. STOP
- b. Wait
- c. Take gingerbread out of oven
- d. Put cake in oven

5. The following program is supposed to drain water from a washing machine. What error does it have?

```

Turn on drainage pump
    
```

```
Check water level
While there is no water left
    Wait
Turn off drainage pump.
STOP
```

- a. Missing instruction
- b. Incorrect sequence
- c. Incorrect decision outcome
- d. Incorrect indentation

LESSON 4 - GETTING STARTED

After completing this lesson, you should be able to:

- Start and run a program
- Enter code
- Create and save a program
- Open and run a program

4.1 INTRODUCING PYTHON



Concepts

This course uses Python, a flexible and widely used computer language that can create simple or complex programs. Python runs faster than most programming languages as there is no delay between requesting the program to run and it actually starting. Also, Python provides a powerful set of **commands**, the instructions of any computer language.

These learning materials use the Python IDLE programming environment, which allows you to write, edit and run code, and save your programs as files for retrieving later. It incorporates the command line environment, the Python Shell.

Python is called an interactive programming language, which means that when you enter a command the system evaluates the command, obeys it and prints or displays the result.

In programming, a prompt appears on the screen to show that the computer is ready to accept inputs. The Python prompt consists of the three characters `>>>`. When this appears, you can start to type your code.

4.2 EXPLORING PYTHON



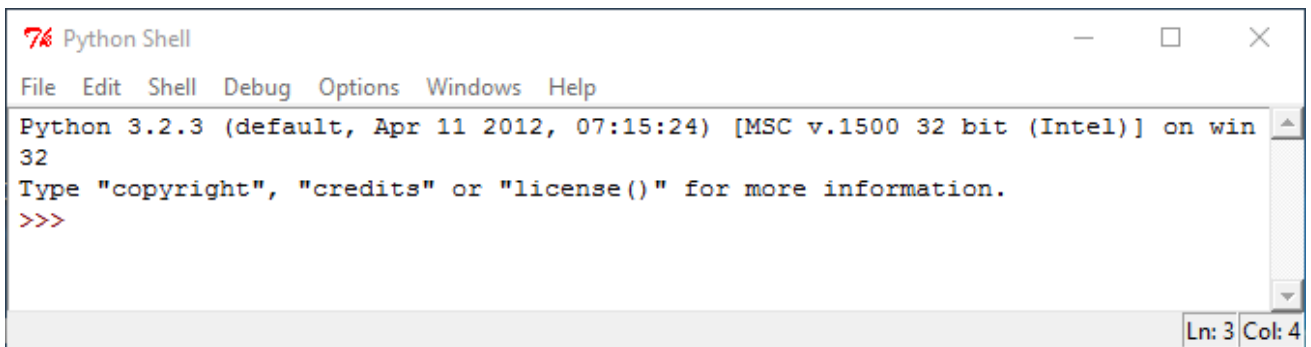
Concepts

Launch Python

1. Locate the Python Shell icon on the computer desktop, or in your program files.

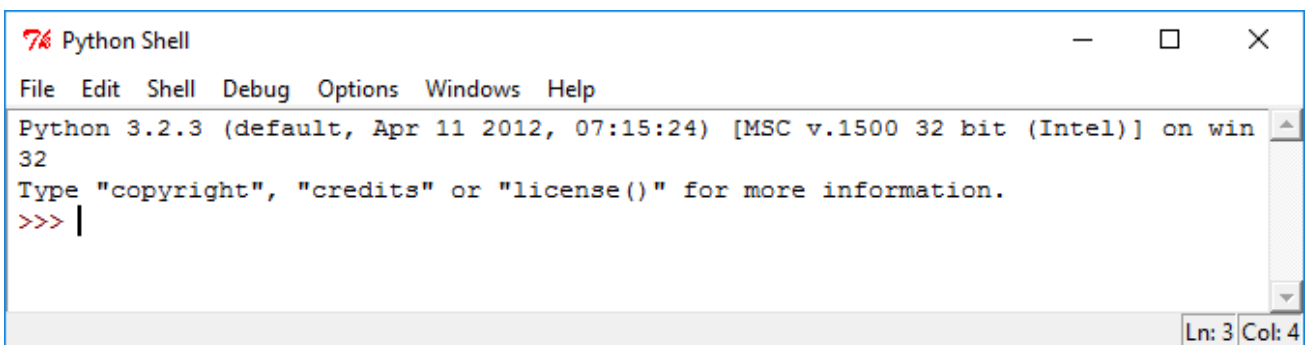


2. Double click on the icon. This will start an interactive window, as shown below:



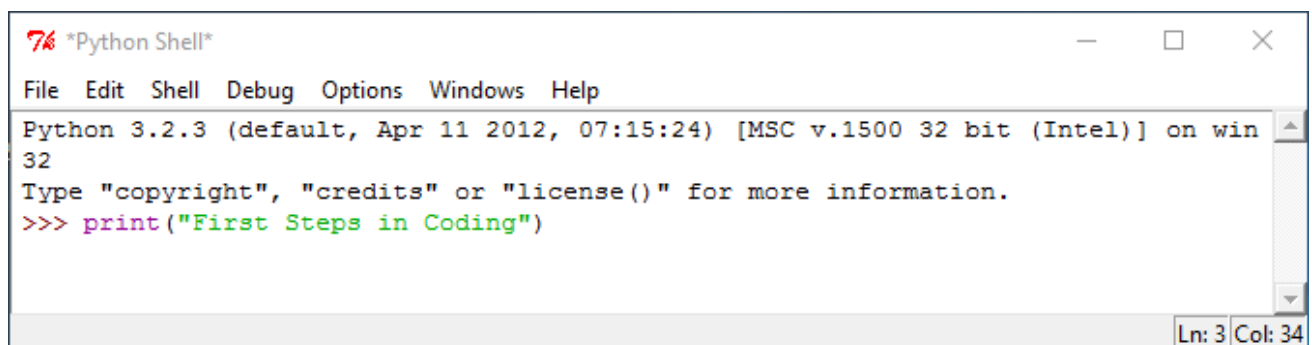
```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>>
```

3. Click the mouse cursor just after the prompt `>>>`. This is the signal to enter code or instructions for Python to run on the computer.



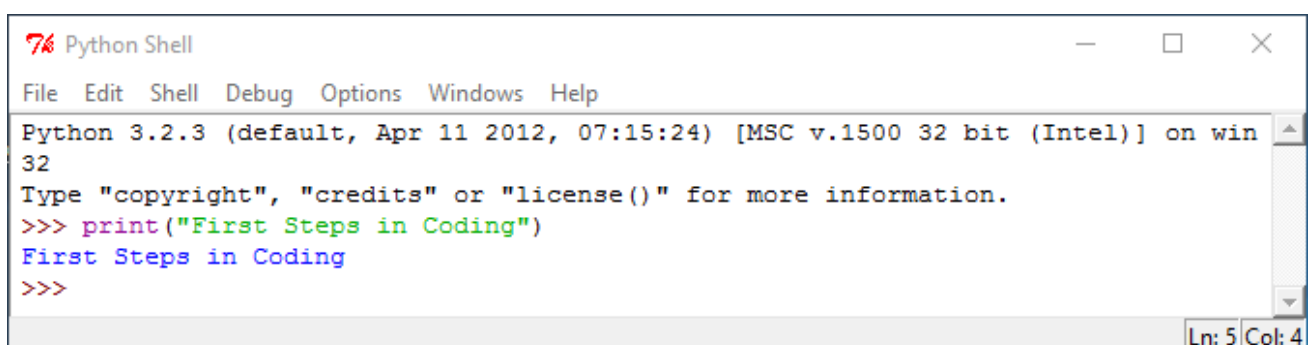
```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> |
```

4. Type `print("First Steps in Coding")`



```
*Python Shell*
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print("First Steps in Coding")
```

5. Press the **Enter** key.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print("First Steps in Coding")
First Steps in Coding
>>>
```

6. View the result.

7. Click **X** to close this window.

How the program works

- In the previous example, Python understood and acted on the code which instructs it to display the text inside the inverted commas onto the screen.

- The result is that the words **First Steps in Coding** are displayed on screen.

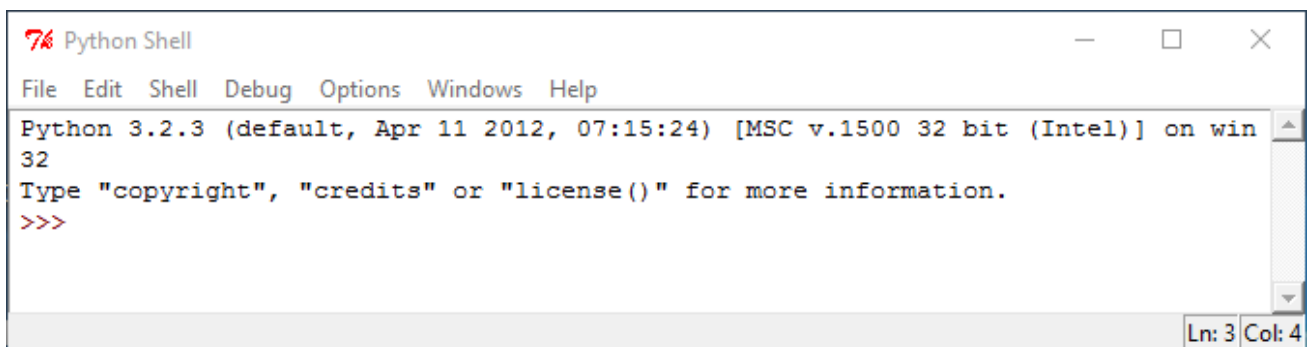
- In Python, the print() command is used to output information to the screen.

Let’s try another instruction:

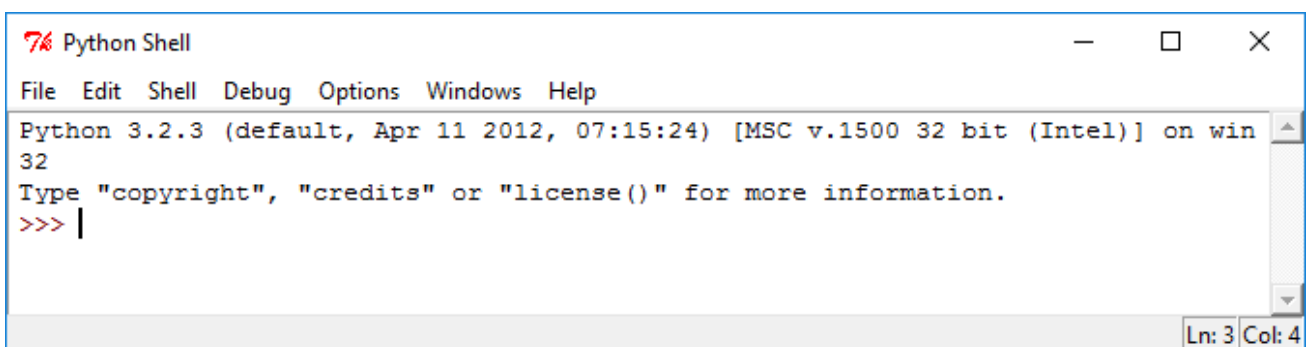
1. Locate the Python Shell icon on the computer desktop, or in your program files.

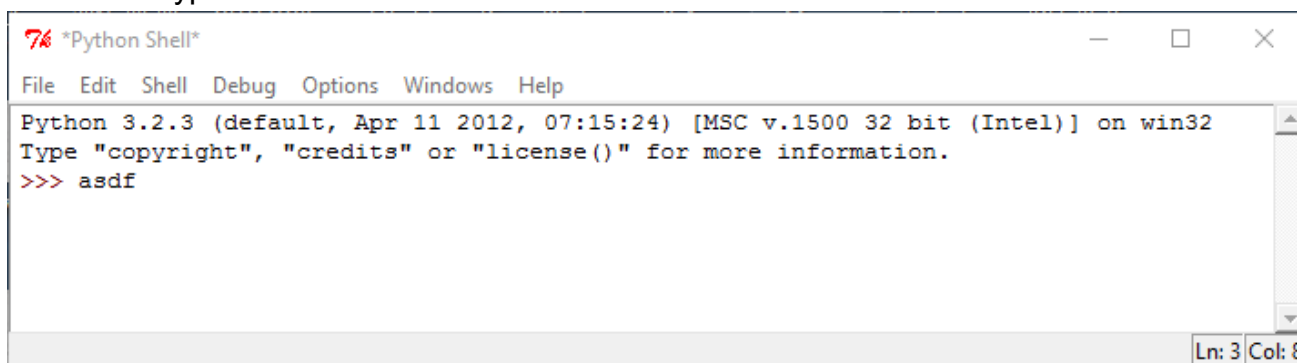


2. Double click on the icon. This will start the interactive Shell window, as shown below:

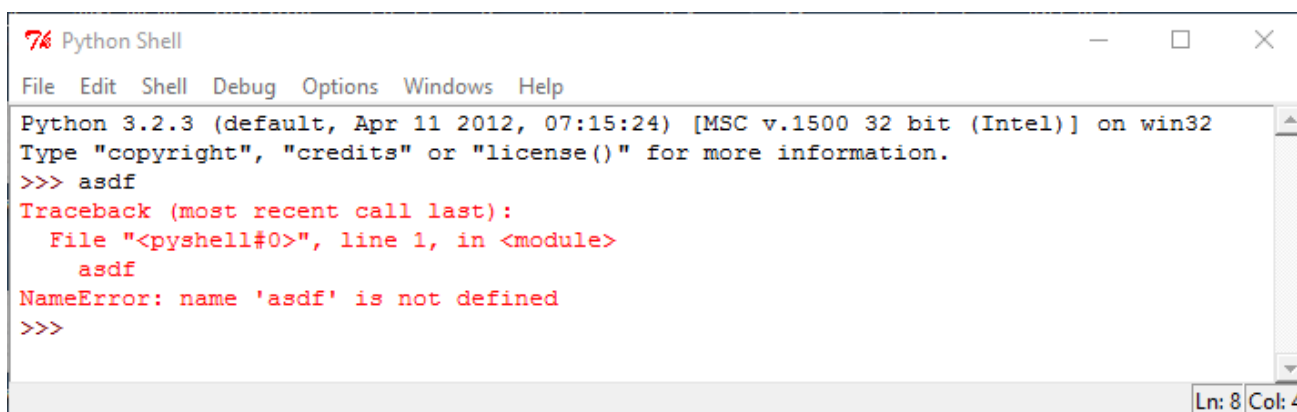


3. Click the mouse cursor just after the prompt >>>. This is the signal to enter code or instructions for Python to run on the computer.



4. Type **asdf**


```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> asdf
Ln: 3 Col: 8
```

5. Press **Enter**


```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> asdf
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    asdf
NameError: name 'asdf' is not defined
>>>
Ln: 8 Col: 4
```

- View the result. In this example, Python does not understand the letters **asdf**. Observe the four lines in red. Python provides **error messages** when it doesn't understand what has been typed in. Usually the last line of the error message has the most useful information about the error, in this case " 'asdf' is not defined ".
- Click **X** to close this window.

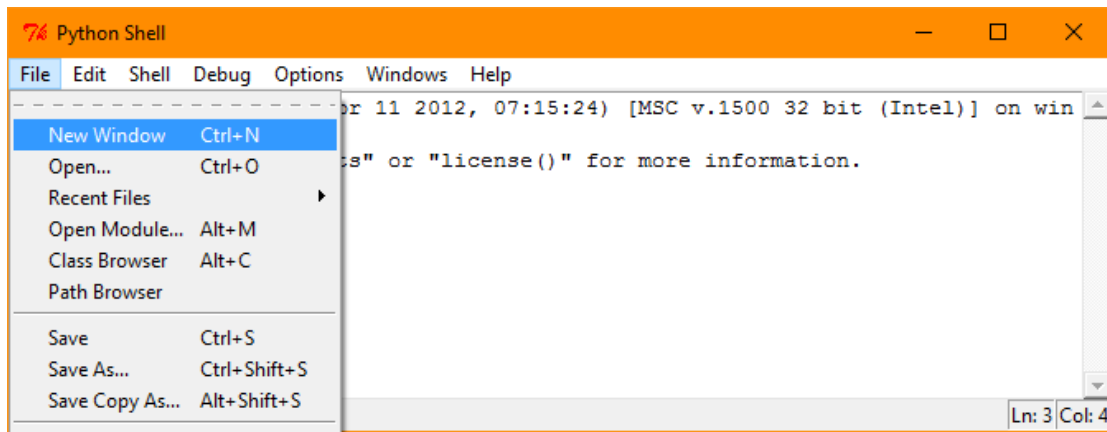
4.3 SAVING A PROGRAM

Concepts

When you write some code, it is good practice to save it so that you can use it again. Python programs have the file extension **.py**. The **.py** extension at the end of the name tells the computer it is a python program, for example, **MagicTrick.py**.

Create and Save a Program

- Open Python. Click **File, New Window** as shown below.

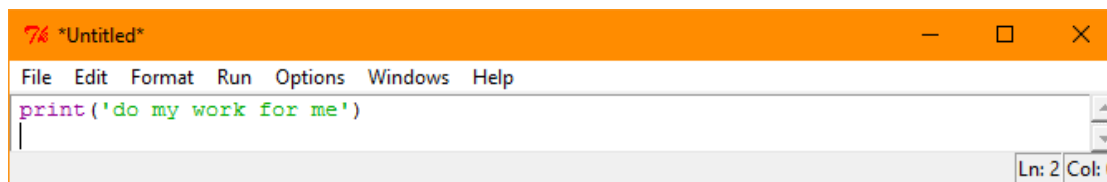


2. A new untitled window is created. Click in the main white area of the untitled window.

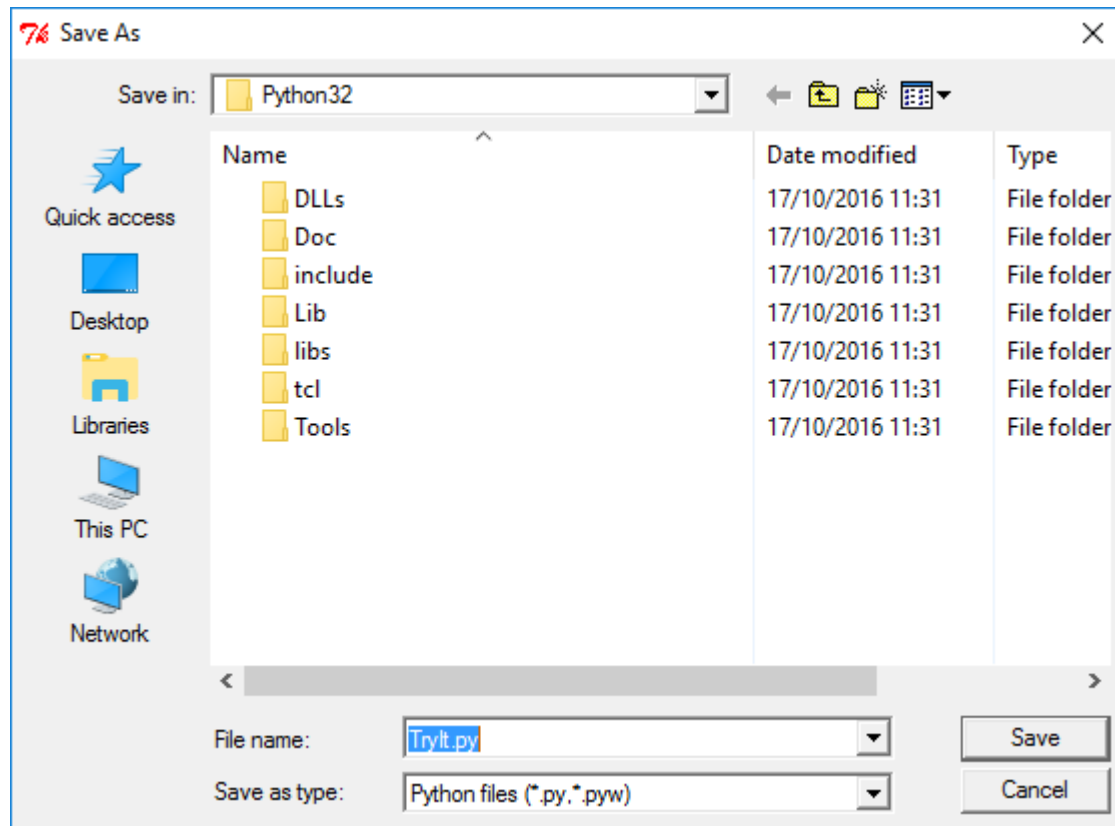


3. Type in the following text and then press enter.

print('do my work for me')

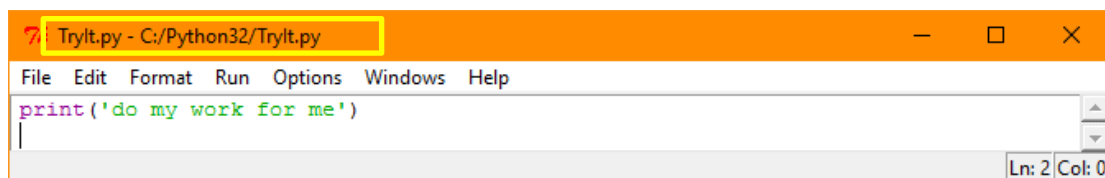


4. Click **File, Save As** and observe the dialogue box. Notice:
 - The default location for saving Python Files is Python 32. This is fine for now, but you can change this depending on the project.
 - The file type displayed is **Python files (*.py,*pyw)**



5. In the file name dialog that appears, type **Trylt.py**.

6. Click the **Save** button.



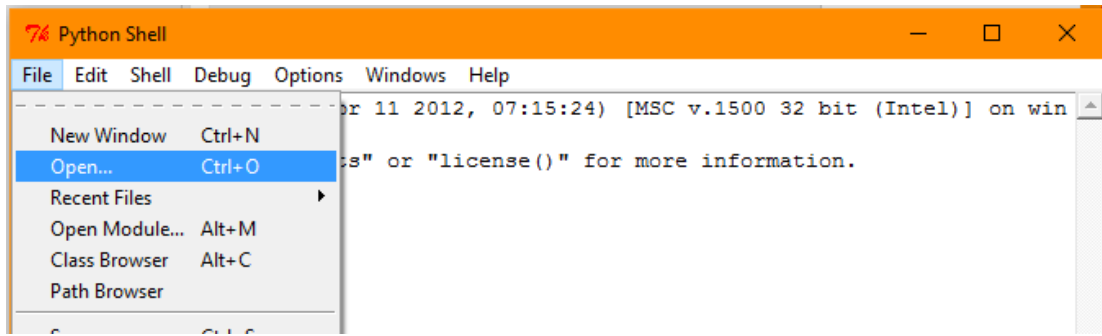
7. Notice the window title changes to '**Trylt.py**'. This is the name of the program or file.

8. Click the X to close the window and the program.

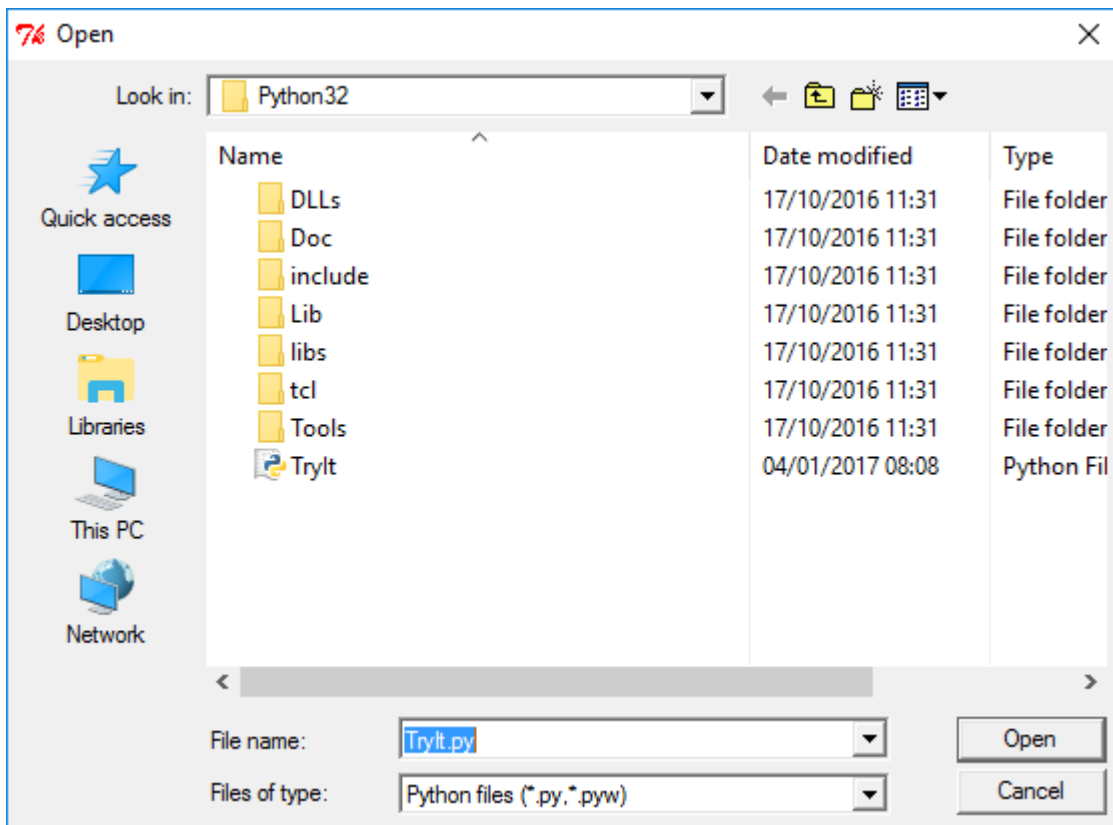
Open and Run an Existing Program

1. Start Python

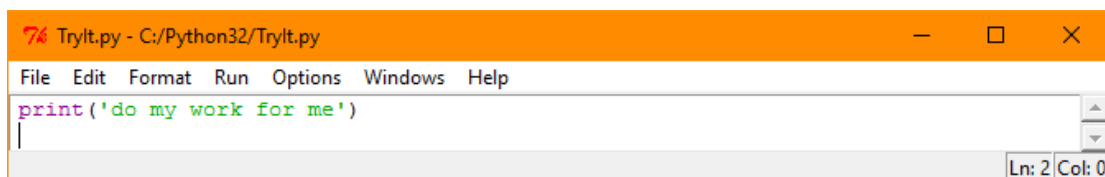
2. Click **File, Open**.



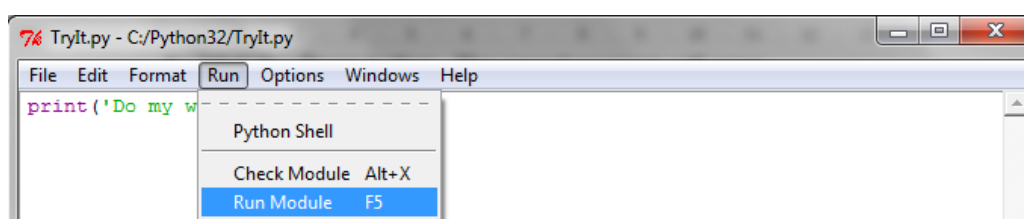
3. In the dialog that appears, select the file name “TryIt.py” or type in the file name.



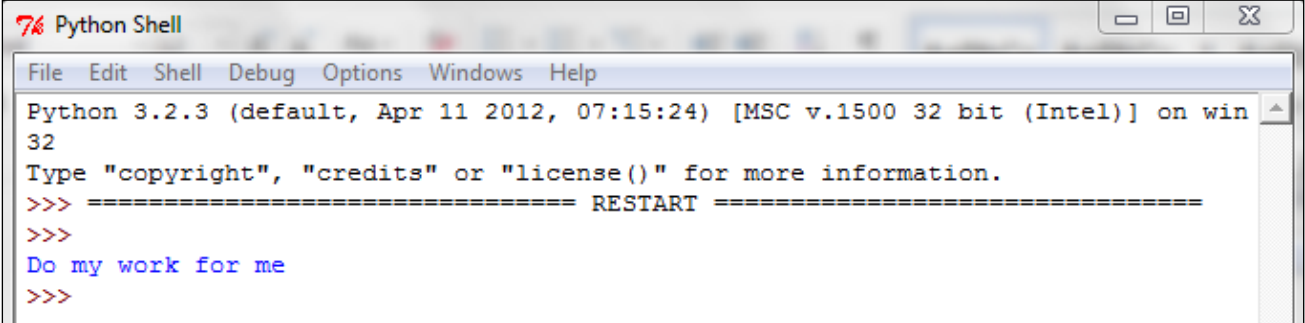
4. Click the **Open** button. The saved program will open.



5. To run the program click on **Run** in the menu bar and then click **Run Module**:



6. This is what you should expect to see when the program is run:

A screenshot of a Python Shell window. The title bar reads "Python Shell" with a red "76" icon on the left and standard window controls on the right. The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area shows the following content:

```
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Do my work for me
>>>
```

7. Pressing the **F5 key** on the keyboard will also run a program. Close the results window by clicking the **X** in the top right corner. Return to your program and this time press **F5** on the keyboard to run the program. View the results.

4.4 REVIEW EXERCISE

1. How do you run a program in Python?
 - a. Type run.
 - b. Press F5.
 - c. Type go.
 - d. Type start.

LESSON 5 - PERFORMING CALCULATIONS

After completing this lesson, you should be able to:

- Recognise and use arithmetic operators. +, -, * and /
- Know how parentheses affect the evaluation of mathematical expressions
- Understand and apply the precedence of operators in complex expressions
- Understand how to use parenthesis to structure complex expressions

5.1 PERFORMING CALCULATIONS WITH PYTHON

Concepts

Operators

In common with most computer languages, Python can perform mathematical calculations, or evaluate mathematical expressions, for example:

$$10+12+15.$$

It uses the symbol `*` for multiply and `/` for divide rather than using `x` and `÷`.

NOTATION	MEANING
<code>3 * 4</code>	3 multiplied by 4
<code>3 / 4</code>	3 divided by 4
<code>3 + 4</code>	3 plus 4
<code>3 - 4</code>	3 minus 4

In Python:

7 multiplied by 9 is written as `7 * 9`
 63 divided by 3 is written as `63 / 3`

The spaces between the numbers are optional and `7*9` will work just as well as `7 * 9`.

The mathematical symbols like `*`, `/`, `+` and `-`, used in programming to perform calculations are called **operators**.

Parentheses Matter

Mathematical expressions in Python can also contain parentheses, also known as brackets, for example:

$$10-(6-4)$$

If we evaluate this expression from left to right, ignoring the parentheses, the solution would be 0.

$$10-(6-4) = 0$$

However, this is incorrect.

Parentheses indicate what should be calculated first. In Python, as in maths, what is inside the parentheses is calculated first. To evaluate an expression like $10-(6-4)$ Python breaks it down as follows:

First: $6-4=2$

Then: $10-2=8$

So, $10-(6-4)=8$

5.2 PRECEDENCE OF OPERATORS



Concepts

When there are parentheses in an expression, it is clear to Python what order to calculate expressions. When there aren't parentheses, there is an accepted order of operations. A rule of formal language determines the order in which the operators are applied. The rule is called the **Precedence of Operators**. In most computer languages the sequence or order in which operators are applied is multiplication, division, addition, subtraction.

Operators $*$ and $/$ are applied before operators $+$ and $-$.

Here are some examples:

$$1+7+2+3$$

$$1+7 = 8, 8+2 = 10, 10+3 = 13 \quad \text{answer: } 13 \quad \text{Correct}$$

$$2*2*3+4$$

$$2*2 = 4, 4*3 = 12, 12+4 = 16 \quad \text{answer: } 16 \quad \text{Correct}$$

$$4+3*2=?$$

You might think the answer should be 14, because:

$$4+3 = 7, 7*2 = 14$$

However, because of precedence of operators, multiplication is done first. $4+3*2$ is calculated exactly as if it had been written $4+(3*2)$. So:

$$4+(3*2) = 4+6, \text{ and } 4+6 = 10 \quad \text{answer: } 10 \quad \text{Correct}$$

$$31*7+112*2$$

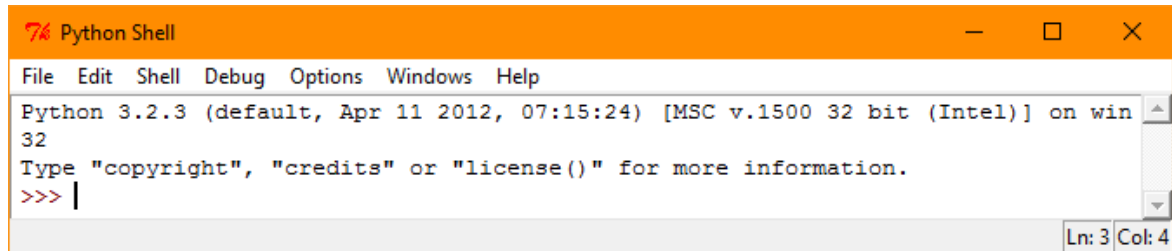
The two multiplication parts are done first, then the addition.

$$31*7+112*2 = 217+224, \text{ and } 217+224 = 441$$

Using Python as a Calculator

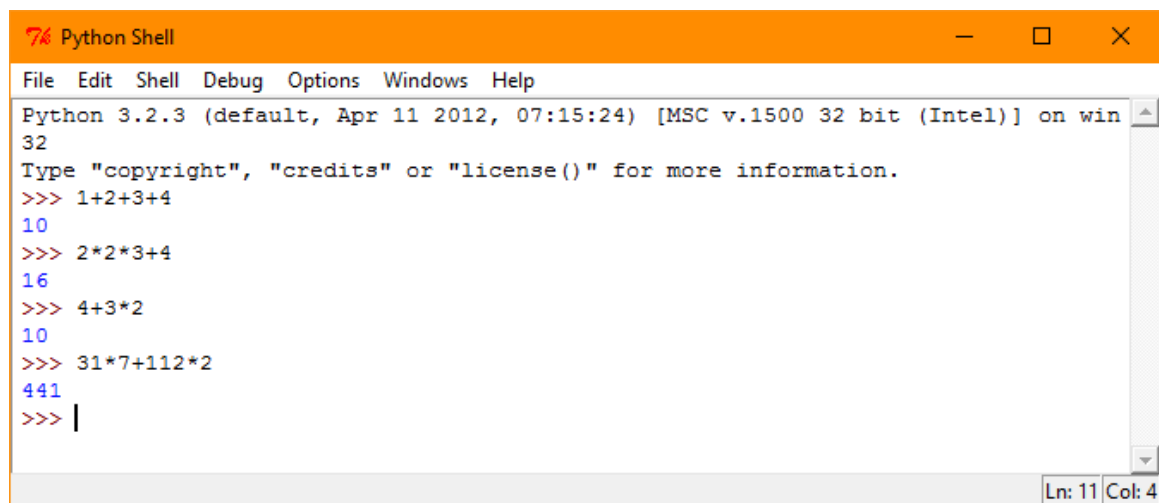
Do the same calculations as in the 'Precedence of Operators' section using Python:

1. Start Python. Position the cursor after the prompt.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> |
```

2. Type in the expressions (calculations) after the prompts. Press **Enter** after typing in each one.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> 1+2+3+4
10
>>> 2*2*3+4
16
>>> 4+3*2
10
>>> 31*7+112*2
441
>>> |
```

3. Check the results.

Example: Calculating the cost of tins of paint

If red paint costs €31 per litre and gold paint costs €112 per litre, then the cost of 7 litres of red paint and 2 litres of gold paint can be written as:

$$31*7+112*2$$

In this case, you can see that it is important to calculate what each colour paint costs first, and then getting the total cost.

So, $(31*7)+(112*2)$ gives the correct overall price, whereas evaluating from left to right does not.

Parentheses are not needed here because of the order of operations.

However it is always possible to put additional parentheses in to override the order of operations. Parentheses have priority over any rules about order of operations.

5.3 REVIEW EXERCISE

1. How do you get Python to evaluate an expression?
 - a. Type 'evaluate', and then type in the expression.
 - b. Break it down into smaller pieces first.
 - c. Type in the expression after the >>> prompt.
 - d. Use a calculator instead.

2. Which of the following expressions is used to multiply two numbers in python?
 - a. $4 + 7$
 - b. 4×7
 - c. $4 \# 7$
 - d. $4 * 7$

3. Evaluate the expression $3*4*2+8$ using the Operator Precedence rule and select the appropriate answer below:
 - a. 24
 - b. 32
 - c. 104
 - d. 120

LESSON 6 – DATA TYPES AND VARIABLES

After completing this lesson, you will be able to:

- Use different data types, character, string, integer, float, Boolean
- Define the programming construct term variable. Outline the purpose of a variable in a program
- Define and initialise a variable
- Assign a value to a variable
- Use data input from a user in a program
- Use data output to a screen in a program

6.1 DATA TYPES



Concepts

In computing all kinds of data are used in order to solve problems and create information. The type of data determines how it is stored in the computer's memory and what can be done with it.

For example, your age is stored as a number and we can perform mathematical calculations on it, like compare it to that of your friend. And your name is stored as text.

In computing data types are used to determine how to store data and what operations to carry out on the data.

The Python **data types** you will use most often are: **Integer**, **float**, **string** and **Boolean**.

Most programming languages have standard data types just with slightly different names. For example the 'string' data type used in Python is sometimes referred to as a 'character' data type in other languages. The character data type can also be used to define letters or characters such as 'a' 'b' 'c' '@' etc.

Data Types in Python

Integer

Describes or defines a whole number of any size, positive or negative, such as 1, 3, 9,999,999, -712 or 0.

Float

Describes or defines a decimal number such as 11.456, -71.3, 29.0.

Floats are also used to represent some very large or small numbers in scientific notation. For example, 5.0e30, which means 5 followed by 30 zeroes, or 1.7e-6 which is 0.000017, which means 1.7 divided by 1 followed by 6 zeros.

String

Describes or defines a piece of text, or 'string' of characters. For example the name of a person, such as 'John Smith'. String values often arise from using the input() command.

Boolean

Describes or defines a value which is either True or False. Boolean data types can only ever be one of these two values. Boolean values most often arise from comparison of numbers.

For example the expression:
 $3 < 4$ gives the boolean value True.

6.2 VARIABLES



Concepts

A **variable** is used in code to represent a piece of data so that the data can be used several times in a program. A variable is like a placeholder for actual values. It can hold the value and then retrieve it for use later.

Assigning a value to a variable

A variable can represent different types of data or values, such as a number or a string. In the code you need to specify what value the variable will represent and this is known as assigning a value to a variable.

In the following example, the value **3** is assigned to the variable named **x**.

x=3

In this case, each time the variable **x** is used in the program it will represent **3**.

A variable can also hold a string, which is a collection of characters. In the following example, the value **Good Morning**, which is a string rather than a number, is assigned to the variable named **Greeting**.

Greeting = 'Good Morning'

In this case, each time the variable **Greeting** is used in the program it will represent **Good Morning**. After the assignment the variable name **Greeting** can be used in place of the string 'Good Morning'.

Using Print() for Output

Python outputs the contents of a variable to the screen, using the **print()** command.

So in the example **x=3**

The command **print(x)** will print the value **3** to the screen.

The command **print(x+2)** will print the value **5** to the screen.

And in the example Greeting = 'Good Morning'

The command **print(Greeting)** will print the value **'Good Morning'** to the screen.

Defining, Initialising and Updating Variables

Defining Variables

Defining a variable means defining in the code what data type the variable will hold, for example, a number (e.g. integer, float) or a string. In Python you do this the first time you assign a value to a variable. Python determines that the variable will hold the data type of the assigned value.

If you assign the number 3 (i.e. the value) to the variable x, Python will assume that x will always represent a numeric value.

x=3

If you assign the string 'country' to the variable q, you are telling Python that q will always represent a string. You use inverted commas around the text to indicate that it is a string data type.

q='country'

In the example, the piece of text is enclosed in inverted commas, which indicates that it is the string data type. Any characters enclosed in inverted commas are considered a string, even a number. In the following example 12 is a string not a number:

Age = '12'

Initialising Variables

You need to assign an initial value to a variable before you can use it. The first time a value is assigned to a variable is called **initialisation of the variable**.

If you try to apply a command to a variable before it has been initialised you will get an error message. Before a value is assigned to a variable, the variable is known as **uninitialised**.

Updating Variables

The value of a variable can be updated by assigning a new value to it. You update the variable and it is automatically updated wherever it is used throughout the program. This is an important reason for using variables as you only have to make one update to update multiple occurrences. A variable always holds the most recent value that has been assigned to it.

Consider a variable called `x` that is currently 3. The assigned value can be changed to 7 by typing the code:

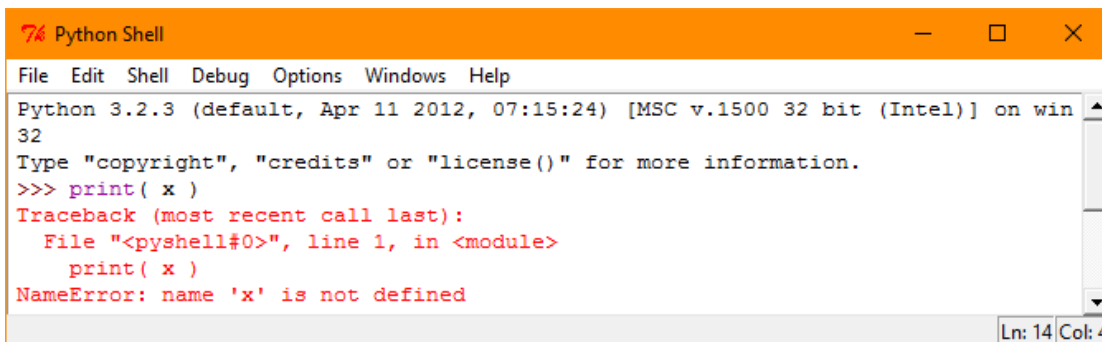
```
x=7
```

The command `print(x)` will now print the value 7.

The command `print(x+2)` will now print the value 9.

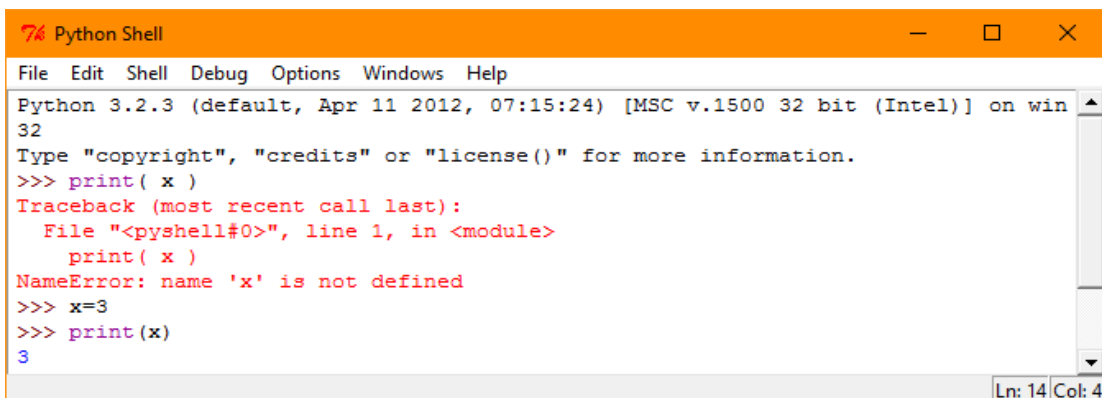
Example: Working with Variables

1. Open Python.
2. Before initialising the variable `x`, type in the command `print(x)`.
3. Observe the error message.



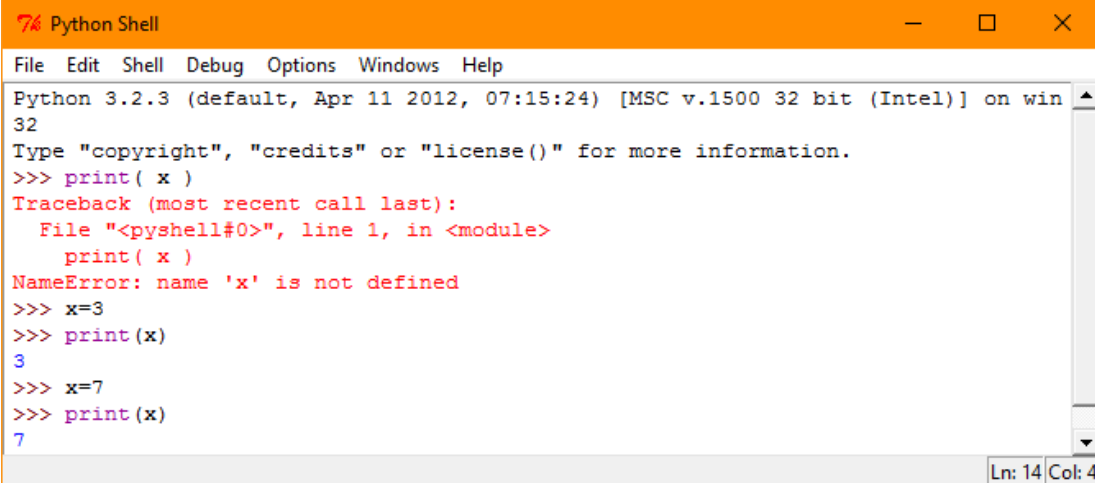
```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print( x )
Traceback (most recent call last):
  File "<pysHELL#0>", line 1, in <module>
    print( x )
NameError: name 'x' is not defined
Ln: 14 Col: 4
```

4. Initialise the variable `x` to 3.
5. Type the print command again.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print( x )
Traceback (most recent call last):
  File "<pysHELL#0>", line 1, in <module>
    print( x )
NameError: name 'x' is not defined
>>> x=3
>>> print(x)
3
Ln: 14 Col: 4
```

6. Update the variable `x` to 7.
7. Print the new value.



```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print( x )
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print( x )
NameError: name 'x' is not defined
>>> x=3
>>> print(x)
3
>>> x=7
>>> print(x)
7
Ln: 14 Col: 4

```

6.3 BEYOND NUMBERS



Concepts

Working with Strings

As well as adding numbers together, you can join strings and characters together. To do this in Python, you can use the + symbol.

For example, initialise the variable **Name** to “David”, and **Question** to “What is your age?”

```
Name= "David"
```

```
Question = "What is your age "+Name+"?"
```

This returns the result:

```
'What is your age David?'
```

Note the blank spaces in the string. Without the spaces at the ends of the string there would be no space between the word you and the word David and the question would look like this:

```
'What is your ageDavid?'
```

Changing strings to numbers

Sometimes a string needs to be converted to a number (e.g. an integer or float). For example if the variable called Age is the string '12',

```
Age = '12'
```

then Age +1 will give an error, because a number can't be added to a string.

To fix that, you can convert the variable Age to an integer as follows:

```
Age = int( Age )
```

Using Input() for User Input

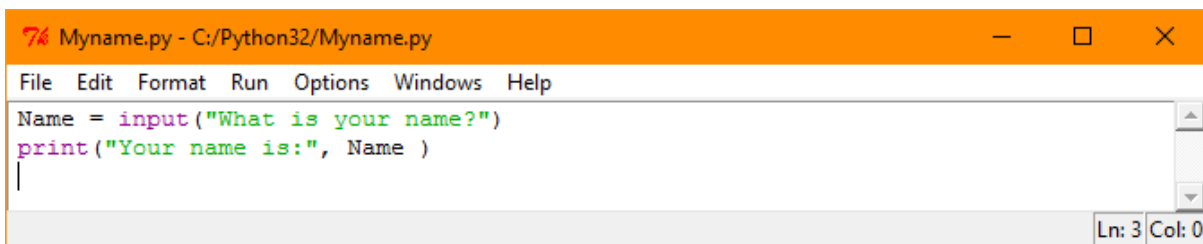
You can write code that asks a user to enter some information in the form of a string, for example their name.

In Python you use the **input()** command to prompt the user to enter a string .

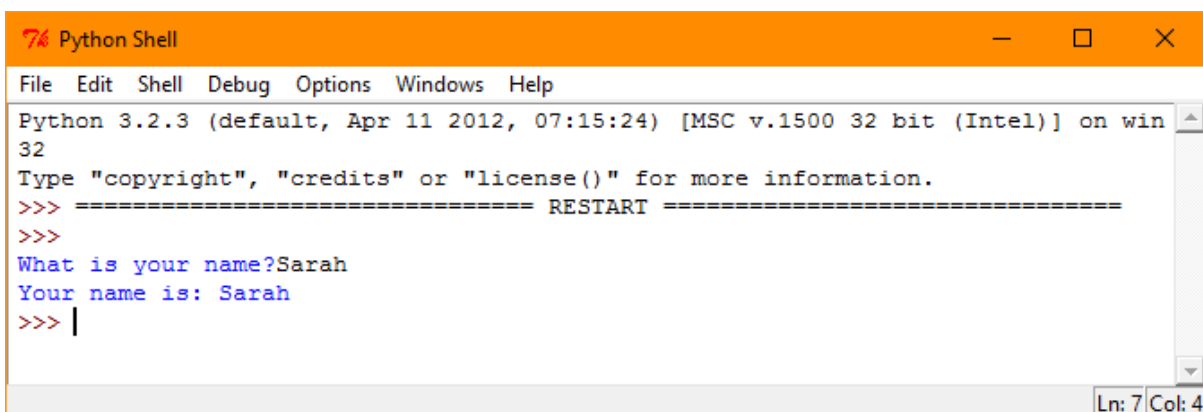
The **input() command** is an instruction in Python that requests the user to enter information and the program waits until the information is entered.

Example: Input and Output a Name.

1. Open Python and create a new file.
2. Type in the following code.



3. Run the program.
4. Reply to the prompt “What is your name?” by typing in your name.
5. Press Enter.
6. Save the program as Myname.py.



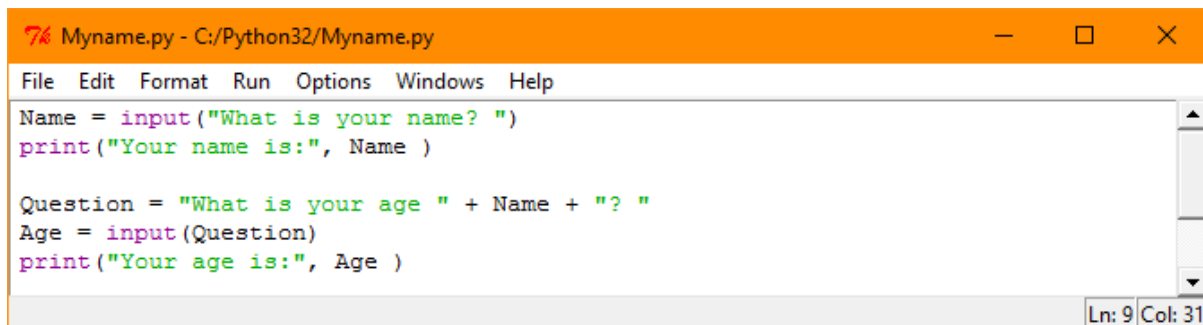
- Click X to close the window.

How the program works

- The user is prompted to enter a string, in this case their name, using the **input()** command.
- The program waits until the information is entered.
- The variable 'Name' is initialised to contain the name entered by the user.
- The **print()** command is used to output the text "Your name is:" followed by the contents of the variable 'Name' to the screen.

Example: Working with Strings

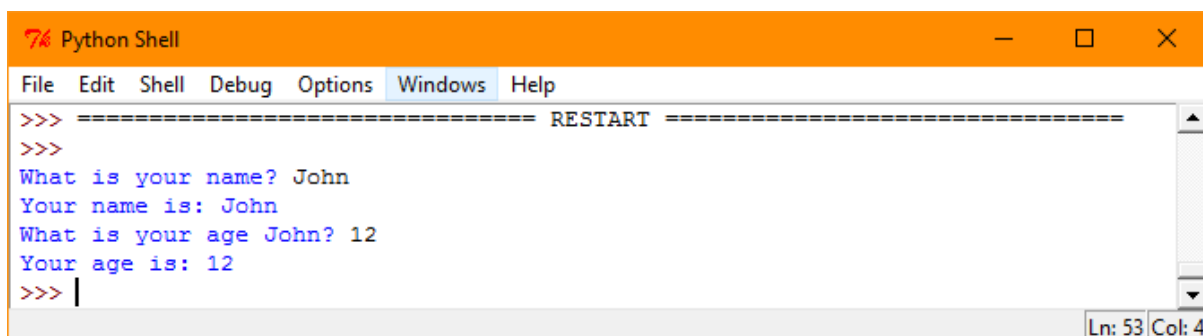
- Open the previous example, MyName.py, using the steps you learned earlier.
- Modify the example so that it looks like this:



```
74 Myname.py - C:/Python32/Myname.py
File Edit Format Run Options Windows Help
Name = input("What is your name? ")
print("Your name is:", Name )

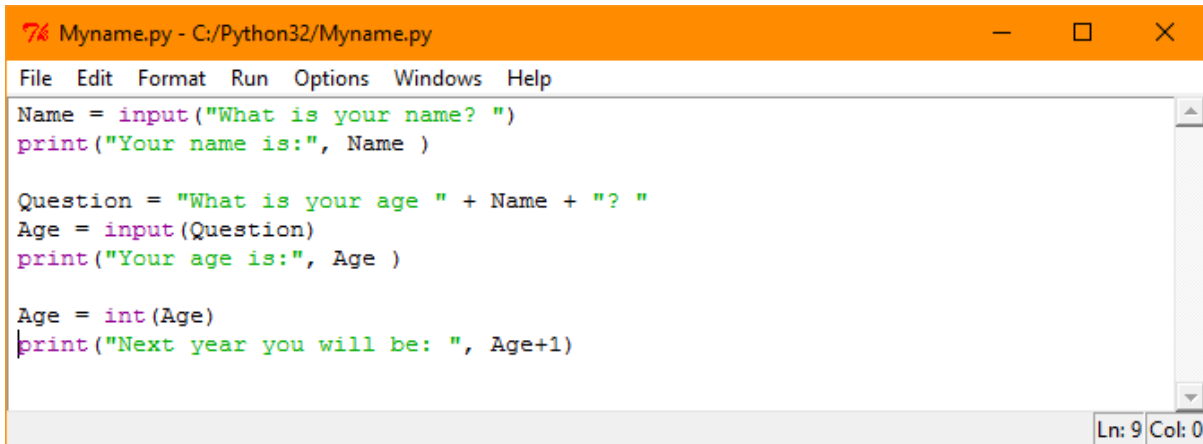
Question = "What is your age " + Name + "? "
Age = input(Question)
print("Your age is:", Age )
Ln: 9 Col: 31
```

- Run the program.
- Answer the first question and press Enter.
- Answer the second question and press Enter.



```
74 Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
What is your name? John
Your name is: John
What is your age John? 12
Your age is: 12
>>> |
Ln: 53 Col: 4
```

6. Modify the program so that it now reads:



```

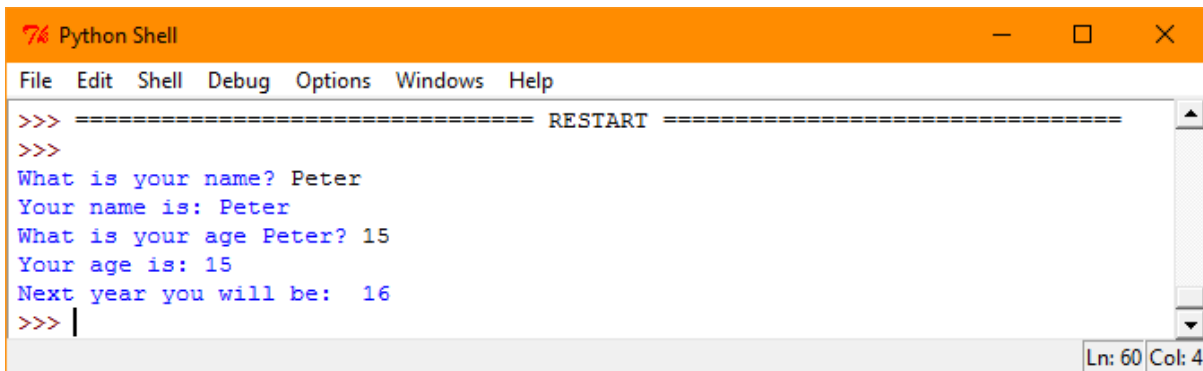
76 Myname.py - C:/Python32/Myname.py
File Edit Format Run Options Windows Help
Name = input("What is your name? ")
print("Your name is:", Name )

Question = "What is your age " + Name + "? "
Age = input(Question)
print("Your age is:", Age )

Age = int(Age)
print("Next year you will be: ", Age+1)
Ln: 9 Col: 0
    
```

7. Now run the program again.

8. Answer the two questions.



```

76 Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
What is your name? Peter
Your name is: Peter
What is your age Peter? 15
Your age is: 15
Next year you will be: 16
>>> |
Ln: 60 Col: 4
    
```

9. Save and close the file.

6.4 REVIEW EXERCISE

1. Which statement about variables is correct?
 - a. Variables never change their value.
 - b. A variable name must use lower case letters only.
 - c. Variables must be initialised to zero.
 - d. Sometimes a variable can hold a Boolean value.

2. What is the purpose of a variable?
 - a. To make it easier to change a program in the future.
 - b. To tell Python how to combine two numbers
 - c. To remember some value that will be used again later in the program.
 - d. To ensure that the source code is indented correctly

3. Which of these Python statements initialises a variable?
 - a. Initialise().
 - b. x=3.
 - c. start().
 - d. finish().

4. In Python, if a variable named 'length' has the value 200, and the instruction 'length=400' is obeyed, what happens?
 - a. Python gives an error message, because you can't change the value of a variable.
 - b. Python gives an error message, because you can't change 200 to 400.
 - c. Python assigns 400 to length and length now has the value 400.
 - d. Python adds 400 into length and length now has the value 600.

5. In Python, the input() function:
 - a. Is used when you want to input a USB key
 - b. Can be used with a prompt to make input faster, or without a prompt to make input slower.
 - c. Returns a string.
 - d. Must be the first function you call in a program.

6. In Python, the output() function
 - a. Is used when you want to change the data type
 - b. Is used to display a result
 - c. Is the last command in a program.
 - d. Must be the first function you call in a program.

7. A data type which has only two values is called
 - a. Integer

- b. Float
- c. String
- d. Boolean

LESSON 7 – TRUE OR FALSE

After completing this lesson, you should be able to:

- Use Boolean logic expressions in a program
- Recognise types of Boolean logic expressions to generate a true or false value like: =, >, <, >=, <=, <>, !=, ==, AND, OR, NOT
- Understand the precedence of operators and the order of evaluation in complex expressions

7.1 BOOLEAN EXPRESSIONS



Concepts

In everyday life, you often check if something is **True** or **False** before deciding what to do next. This concept is also used in computer programs where conditions are checked to see if they are **True** or **False** before decisions are made on what to do next. For example:

Is it cold outside?

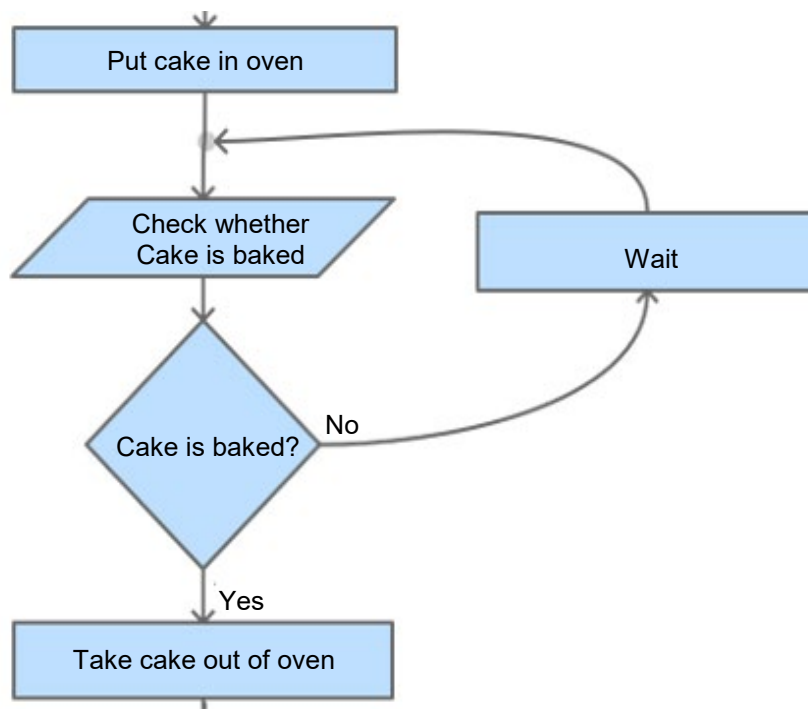
True – wear a coat

False – leave the coat at home

Is the cake baked?

True – take it out of the oven

False – bake it for 5 more minutes and re-check



True or False in Flowcharts

In a flowchart the checks or tests are represented by decision boxes with outcomes **Yes** or **No**.

Computer programs work with the outcomes **True** and **False** rather than Yes or No.

True and False are important concepts in computing because we need a way to test if certain conditions exist in order to control what happens next in the program.

In computing, this logic test is called a **Boolean expression**. A Boolean expression results in a **Boolean value** that is either true or false.

One way in which Python programs use Boolean expressions is to test the size of a numerical value against another value and give a 'True' or 'False' result. It carries out this test by using a **comparison operator**. A comparison operator compares the values on either side of them and decide the relation among them. They are sometimes referred to as relational operators.

$$99 > 7$$

Here the operator '>' compares the values 99 and 7 to see if the value on the left is greater than the value on the right. If so, the result is True, otherwise the result is False.

So: $99 > 7$ is True but $6 > 7$ is False.

7.2 COMPARISON OPERATORS

Concepts

Comparison operators are a type of Boolean logic expression used to compare values and decide if the result is true or false.

Table of Comparison Operators

There are six comparison operators. In the table below X and Y are variables holding numerical values:

NOTATION	MEANING
$X > Y$	X is greater than Y
$X < Y$	X is less than Y
$X \geq Y$	X is greater than or equal to Y
$X \leq Y$	X is less than or equal to Y
$X == Y$	X is equal to Y

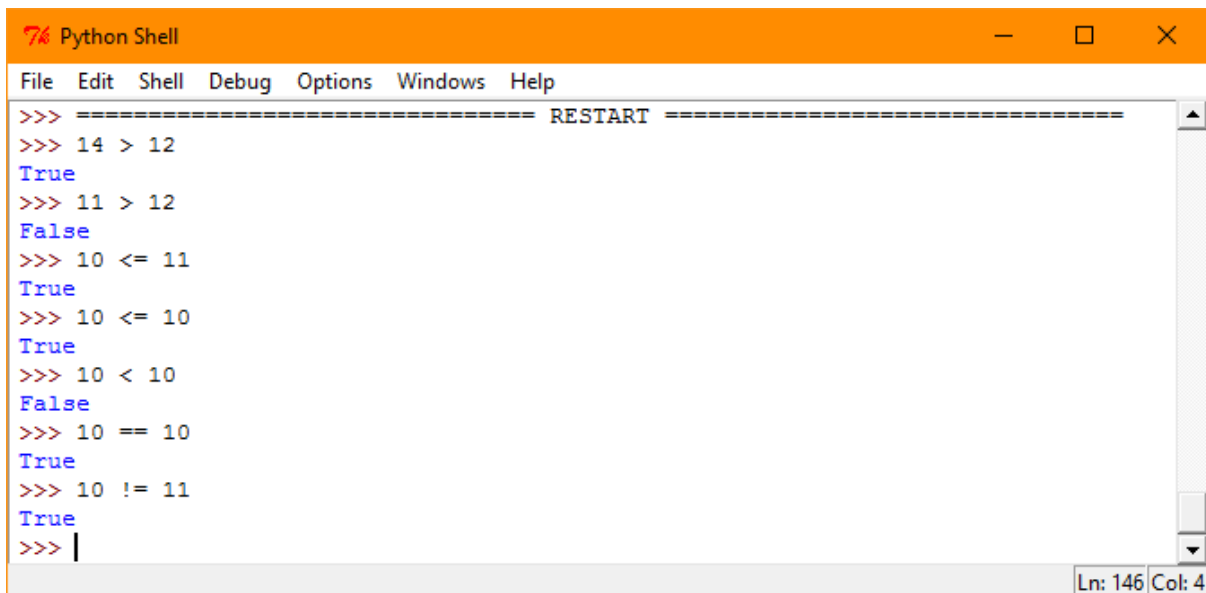
$X \neq Y$

X is not equal to Y

Note: Some other languages may use $\lt\gt$ as not equal to and $=$ as equal to. Python 3.x doesn't use $\lt\gt$ and it uses $==$ instead of a single $=$.

Example: Comparison Operators

1. Open Python.
2. Enter the examples below to check your understanding of the comparison operators.



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>> 14 > 12
True
>>> 11 > 12
False
>>> 10 <= 11
True
>>> 10 <= 10
True
>>> 10 < 10
False
>>> 10 == 10
True
>>> 10 != 11
True
>>> |
```

3. Make up your own examples. Anticipate what the answer to each should be. See whether the answers, True or False, are what you expect them to be.

7.3 BOOLEAN OPERATORS



Concepts

Boolean operators are the words **and**, **or**, **not** which are another Boolean logic expression used to combine Boolean values like True and False together to give a final result.

There are three basic Boolean operators:

and

- Combines two Boolean values and gives a result - True if both values are True, otherwise False

<i>Expression</i>	<i>Result</i>
True and True	True
True and False	False
False and True	False
False and False	False

or

Combines two Boolean values and gives a result - True if either one or both values are True, otherwise False.

<i>Expression</i>	<i>Result</i>
True or True	True
True or False	True
False or True	True
False or False	False

not

Converts a single Boolean value from True to False, or False to True.

<i>Expression</i>	<i>Result</i>
not True	False
not False	True

Boolean expressions can include the use of both comparison operators and Boolean operators. This is an example:

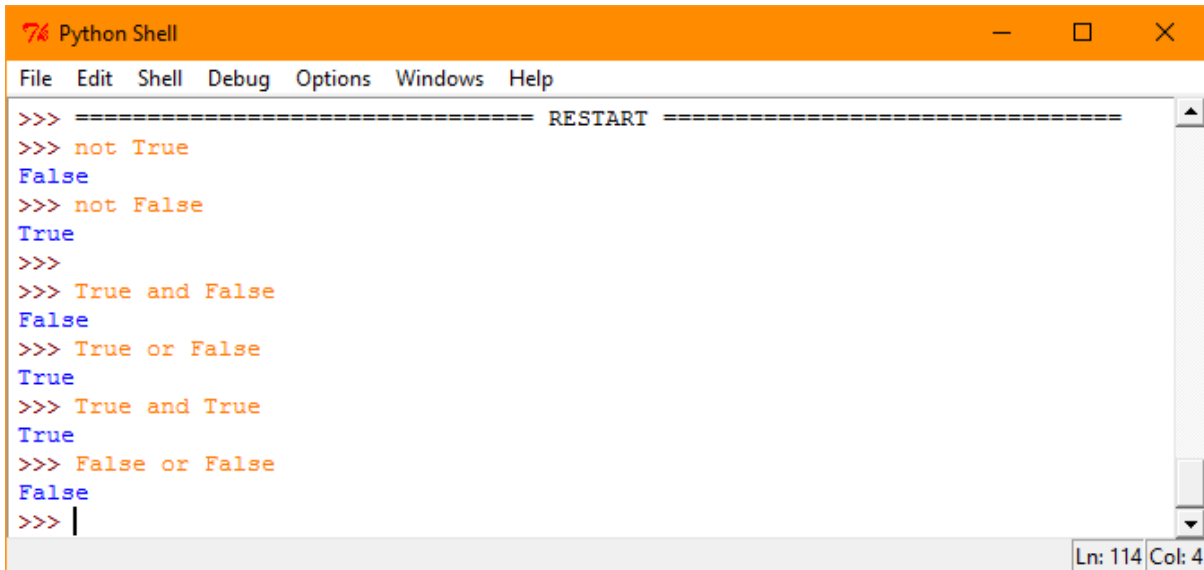
$$x < 3 \text{ and } y < 12$$

In such mixed expressions, the comparison operators are evaluated before the Boolean operators. That's because of the rules of operator precedence which we'll come back to later in this lesson.

Example: Boolean Expressions

1. Open Python.

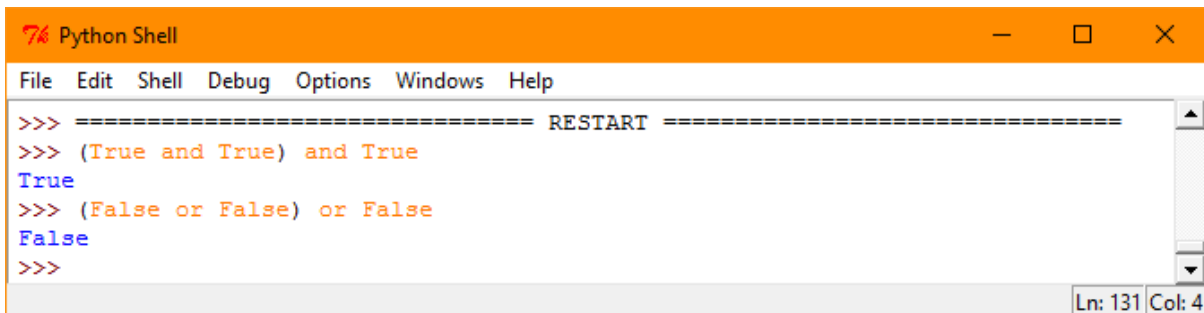
2. Try to evaluate Boolean expressions in Python.
3. Think of some combinations to make sure that you understand what **and**, **or** and **not** do.



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>> not True
False
>>> not False
True
>>>
>>> True and False
False
>>> True or False
True
>>> True and True
True
>>> False or False
False
>>> |
```

Ln: 114 Col: 4

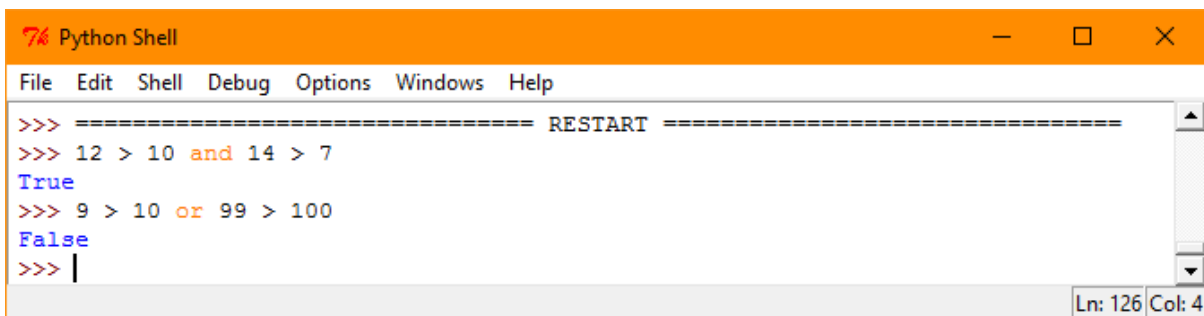
4. Build more complex Boolean expressions with parentheses, remembering that the expression within the parentheses is worked out first.



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>> (True and True) and True
True
>>> (False or False) or False
False
>>> |
```

Ln: 131 Col: 4

5. Use Python to evaluate some Boolean expressions that arise from comparing numbers.



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>> 12 > 10 and 14 > 7
True
>>> 9 > 10 or 99 > 100
False
>>> |
```

Ln: 126 Col: 4

7.4 BOOLEANS AND VARIABLES



Concepts

Initialising and assigning values to variables applies to numbers, strings and Booleans.

Consider the instruction:

```
Y = 200
```

This initialises the variable called Y to the numeric value 200.

Consider the instruction:

```
A = False
```

This initialises the variable called A to the Boolean value False.

Consider the instruction:

```
name=input("What is your name?")
```

This initialises the variable called name to a string in response to the question, "What is your name?"

Here is what happens:

- i. The user is prompted with the words "What is your name?"
- ii. The user types in a string.
- iii. The string value is assigned to the variable called name.

The next example uses a Boolean variable. The variable is called True_or_false.

Notice that underscores are being used between the words of the variable name. Python treats an underscore within a variable name like other letters, so True_or_false is one variable. If you use spaces instead of underscores, Python will treat the words as three separate things and give an error.

Consider the instruction:

```
True_or_false = Age > 12
```

It compares Age with 12 and puts the Boolean result into the variable True_or_false.

Example: Working with Booleans

1. Open the program Myname.py, and modify it as shown below.

```

7% Myname.py - C:/Python32/Myname.py
File Edit Format Run Options Windows Help
Name = input("What is your name? ")
print("Your name is:", Name )

Question = "What is your age " + Name + "? "
Age = input(Question)
print("Your age is:", Age )

Age = int(Age)
print("Next year you will be: ", Age+1)

True_or_false = Age > 12
print( Name, "is older than 12 is", True_or_false )
Ln: 12 Col: 33
    
```

2. Run the program.
Type in a name.
Type in some age greater than 12.

```

7% Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
What is your name? Sue
Your name is: Sue
What is your age Sue? 14
Your age is: 14
Next year you will be: 15
Sue is older than 12 is True
>>> |
Ln: 84 Col: 4
    
```

3. Run the program again.
Type in a name.
Type in some age less than 12 to see a different result.

```

7% Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
What is your name? Dave
Your name is: Dave
What is your age Dave? 11
Your age is: 11
Next year you will be: 12
Dave is older than 12 is False
>>> |
Ln: 92 Col: 4
    
```

7.5 PUTTING IT ALL TOGETHER

Concepts

Parentheses in Boolean Expressions

We have seen in earlier lessons that parentheses are important in numerical expressions. They are also important in Boolean expressions.

(A and B) or C is not the same thing as **A and (B or C)**

Let's assume for example that A is False, B is True and C is True. Putting these values into the two expressions gives different results:

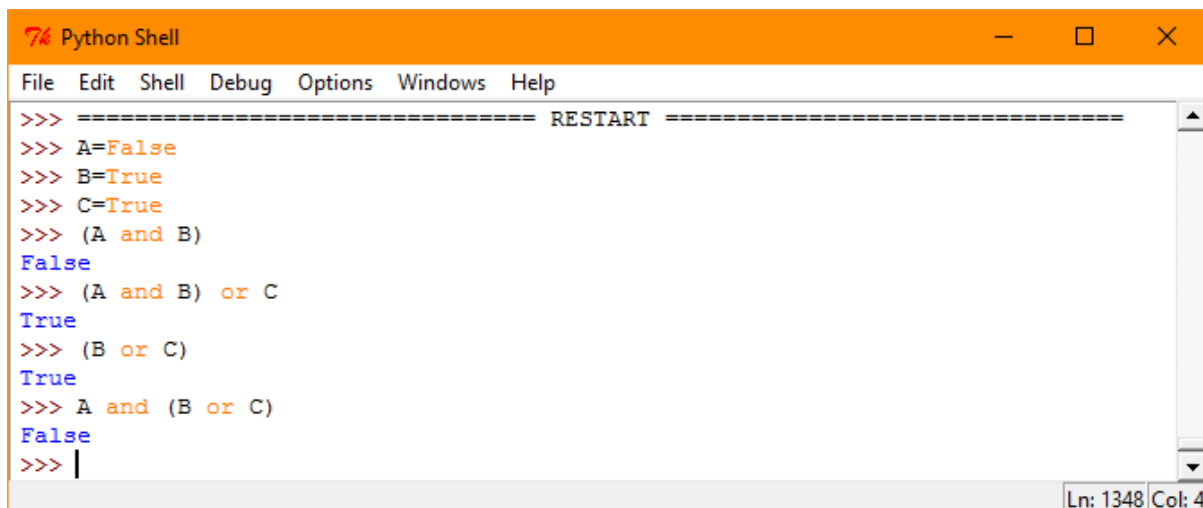
(A and B) or C evaluates to True

A and (B or C) evaluates to False

To confirm this, type in both Boolean expressions into Python, using it as a calculator as shown in the next example.

Example: Parentheses in Boolean Expressions

1. Open Python. Type in the following expressions:



```
>>> ===== RESTART =====
>>> A=False
>>> B=True
>>> C=True
>>> (A and B)
False
>>> (A and B) or C
True
>>> (B or C)
True
>>> A and (B or C)
False
>>> |
```

Note how the use of parentheses changes the result of the expression A and B or C.

Operator Precedence, including Booleans

The table below is a partial list of the order in which the most important operators are applied, or take precedence. In the list below, the operators at the top of the list are applied before operators at the end.

<i>Symbol</i>	<i>Operation</i>
<i>* /</i>	Multiply and divide
<i>+ -</i>	Addition and subtraction
<i><, >, <=, >=, ==, !=,</i>	Comparison operators
<i>not</i>	Boolean not
<i>and</i>	Boolean and
<i>or</i>	Boolean or

For example, in the absence of parentheses *** is evaluated before *+*, as *** is before *+* in the list. Likewise, '**and**' is evaluated before '**or**', as '**and**' is before '**or**' in the list.

An expression like:

A or B and C

Will be evaluated as:

A or (B and C)

A complex expression such as:

$x < 2$ or $1+3*x > 2+4/2$ and not $x < y$

Would be calculated as:

$(x < 2)$ or $(1+(3*x) > 2+(4/2)$ and $(not (x < y))$)

Even experienced programmers sometimes forget exactly what order the operators are evaluated in. In an expression as complex as the above it is usual to include some parentheses to make the intention clearer:

$x < 2$ or $((1+3*x > 2+4/2)$ and not $x < y)$

The operator precedences for ***, */*, *+* and *-* are used so frequently that it is rare to use parentheses to clarify the order of evaluation.

7.6 REVIEW EXERCISE

1. Which of the following is a python expression to add three numbers together and divide the result by 3?
 - a. $4+5+6/3$
 - b. $4+5+6\div 3$
 - c. $(4+5+6)/3$
 - d. $(4+5+6)\div 3$

2. The expression ' $3 < x$ and $x \leq 7$ ' is true if x is:
 - a. 3,4,5,6 or 7
 - b. 2,3,4,5 or 6
 - c. 4,5, or 6
 - d. 4,5,6 or 7

3. In Python, which expression could give a different result to the other three?
 - a. $a < b < c$
 - b. $c > b > a$
 - c. $a < b$ or $b < c$
 - d. $a < b$ and $b < c$

LESSON 8 – AGGREGATE DATA TYPES

After completing this lesson, you should be able to:

- Understand aggregate data types
- Use aggregate data in a program like: list and tuple

8.1 AGGREGATE DATA TYPES IN PYTHON



Concepts

As well as standard data types such as int, string and Boolean, there is another category of data type called aggregate data types. Aggregate data types hold multiple items, for example, a list of names. Most programming languages deal with aggregate data types in some way for example, an array contains a group of elements of the same data type.

Two Python aggregate data types are **list** and **tuple**.

List

A collection of values. Lists can change by adding items to or removing items from them or changing an item in a list. Python can display an entire list of values. It can also retrieve items from a list. (The Python 'list' aggregate data type fulfils the role of an 'array' in other programming languages.)

Tuple

A tuple is just like a list, except once it has been created it can't be changed by rearranging, adding or removing items or modifying individual items.

While the restriction on tuples might appear to make them less useful than lists, tuples have two important advantages:

1. Retrieving items from Tuples is faster than retrieving items from lists.
2. A programmer reading code for tuples will not need to check if values were changed.

The items in a list or tuple are called **elements**.

An element in a list or tuple can be selected by giving the name of the list or tuple and then the number of the element in square brackets. For example, element 3 of the list called pets is:

```
pets[3]
```

Elements are referenced by numbers starting from zero, so in the list:

```
["cat", "dog", "hamster", "goldfish", "rabbit", "turtle"]
```

```
pets[0] = cat
```

```
pets[1] = dog
```

```
pets[3] = goldfish
```

8.2 LISTS

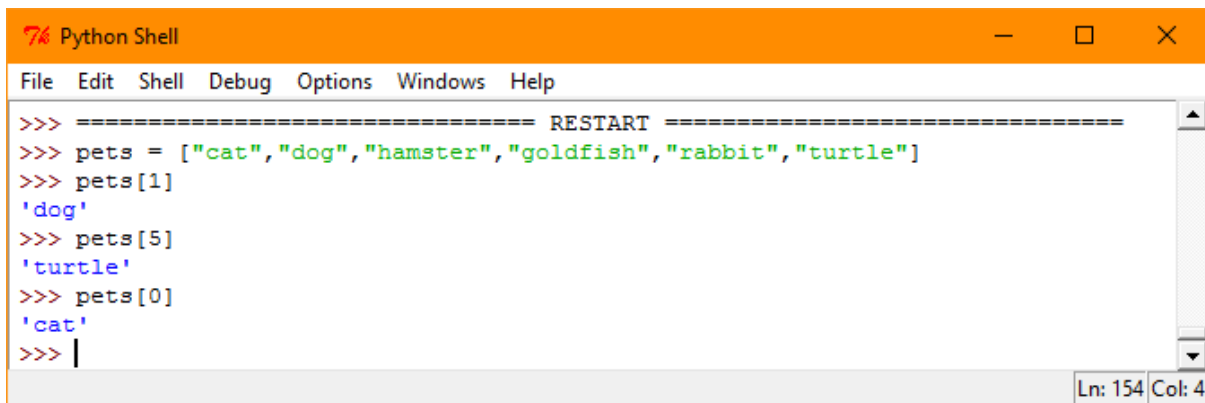
Concepts

Python has some built-in methods for working with lists. For example, this command will sort the list called pets:

```
pets.sort()
```

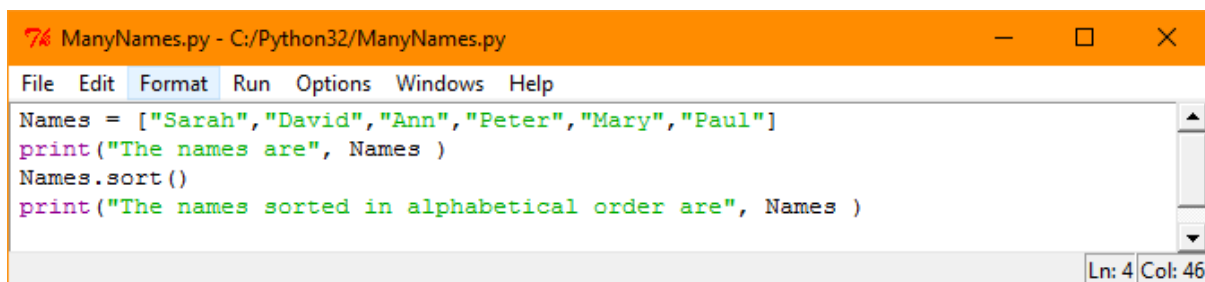
Example: Working with Lists

1. Open Python.
2. Create a list called pets, by enclosing the elements in square brackets, as you see in the screenshot below.
3. Explore individual elements of the list by using numbers in square brackets as shown.

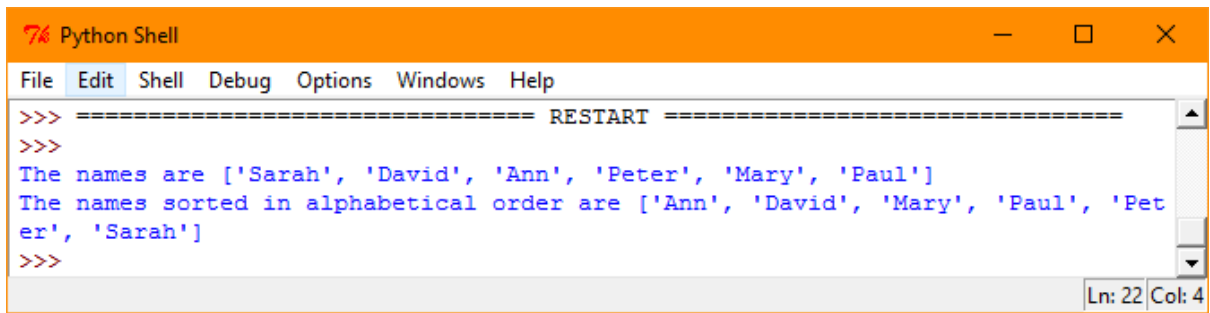


Example: Sorting a List in Python

1. Open Python.
2. Create a program file called ManyNames.py.
3. Type in the program code as below, using the names of some friends if you like, rather than the names given here.



4. Run the program.



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
The names are ['Sarah', 'David', 'Ann', 'Peter', 'Mary', 'Paul']
The names sorted in alphabetical order are ['Ann', 'David', 'Mary', 'Paul', 'Peter', 'Sarah']
>>>
```

8.3 TUPLES



Concepts

Tuples in Python

Tuples are created in the same way as lists, but using round brackets instead of square brackets.

You cannot sort a tuple with the **sort()** command, as Python does not allow rearrangement of a tuple's elements.

If a variable is assigned to a tuple, Python still can't make changes to that tuple. Python can however assign a new variable to the tuple i.e. replace the tuple with a new one. This sequence shows an example of how you might replace one tuple with a new one:

```
animal_tuple = ('cat', 'dog', 'pig')
animal_tuple = ('elephant', 'tiger', 'giraffe')
```

The second tuple replaces the first.

One common use for a tuple is to hold the x and y position of a point on the screen. Instead of having two variables, one for x, and one for y, a single tuple variable can hold both values. When the x and y values are needed, they can be extracted from the tuple.

(A list could be used in the same way as a tuple to hold the x and y position of a point on the screen but a tuple is more efficient.)

Example: Attempting to Sort a Tuple:

1. Open Python.
2. Use the code shown below to create a tuple containing names of fruit.
3. Show the tuple's value by typing 'fruit'.

4. Attempt to sort the tuple.

```

Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>> fruit = ('apple','orange','banana')
>>> fruit
('apple', 'orange', 'banana')
>>> fruit.sort()
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    fruit.sort()
AttributeError: 'tuple' object has no attribute 'sort'
>>> |
Ln: 175 Col: 4
    
```

Example: Extracting x and y from a Tuple

1. Open Python.
2. Use the code shown to create a tuple containing 40 and 150.
3. Extract x and y from the tuple.
4. Show the value of x and y.

```

Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>> p=(40,150)
>>> x=p[0]
>>> y=p[1]
>>> x
40
>>> y
150
>>> |
Ln: 191 Col: 4
    
```

LESSON 9 – ENHANCE YOUR CODE

After completing this lesson, you should be able to:

- Describe the characteristics of well-structured and documented code like: indentation, appropriate comments, descriptive naming
- Define the programming construct term comment. Outline the purpose of a comment in a program
- Use comments in a program

9.1 READABLE CODE



Concepts

Imagine you have just completed a program for a game you have invented. You know exactly how the code works because you have just written it. You save your work and move on to another project. Now imagine returning to the program after a few months; how quickly would you be able to familiarise yourself with the code? How easy would it be for someone else who has never seen it to figure it out?

In reality python programs can be quite large and contain many instructions. They are often worked on by more than one programmer at different stages. It is essential that a program's code is 'readable' i.e. the reader can easily understand what the program does and why.

The same general techniques for making code easier to understand apply to all programming languages, they are:

- Use of comments
- Organisation of code
- Descriptive names

9.2 COMMENTS



Concepts

A **comment** is a piece of text explaining what some part of the code does. It is a short description of a piece of code that humans can read but the computer will ignore, designed to help programmers understand what is happening within a program. Appropriate comments are a characteristic of well-structured and documented code. Comments on the code should help the author and other people understand what each section of code is doing.

Comments start with a hash, #, and go on to the end of the line. They appear in red in the Python editor.

The red text after the # to the end of the line is not considered by Python when it is running the code. The text is just there for programmers use.

This is a comment

9.3 ORGANISATION OF CODE

Concepts

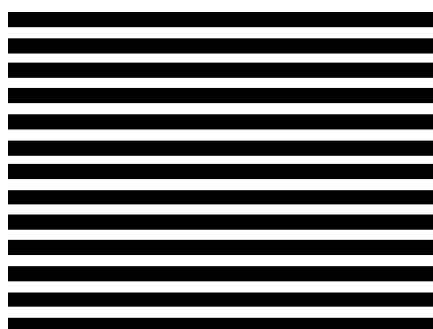
Programmers make their code easier to follow by breaking it up into smaller chunks, or blocks.

A block is defined as a group of adjacent lines that are indented the same amount. Indentation is used to make program code easier to read and understand. Many code editors perform the indentation automatically.

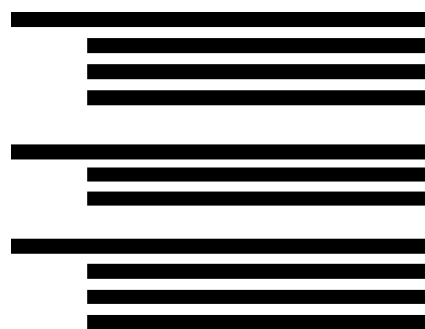
Blocks are usually named by using the word 'def' followed by the name, chosen to describe what the code does. (This is explored further in lesson 12 Procedures and Functions.)

The name (“def name”) is entered first, and then the start of the block is shown with an indentation of the line. The lines of code in a block have the same indentation, and are a sequence, so they are run one after the other.

By using carefully named blocks, a programmer can give a clearer structure to a large Python program.



'Unblocked' code



'Blocked' organisation of code

Figure 1

9.4 DESCRIPTIVE NAMES

Concepts

Descriptive naming is a characteristic of well-structured and documented code. It is the process of giving meaningful names to items and functions and procedures in programming. There is an accepted structure which is common to all languages. For example to write a small function called 'egg timer' in Python, it should be written 'egg_timer'. This allows readers to see what the function is intended to do.

Giving meaningful to variable names will also make more sense to the reader. Variable names should be carefully selected so that they give a clue as to what the variable is used for.

In our list example from lesson 8, the variable name **pets** could have been used instead for a list of fruit:

```
pets=['pineapple','mango','kiwi fruit']
```

Python would have no problem with that, and if instructed:

```
print( pets[1] )
```

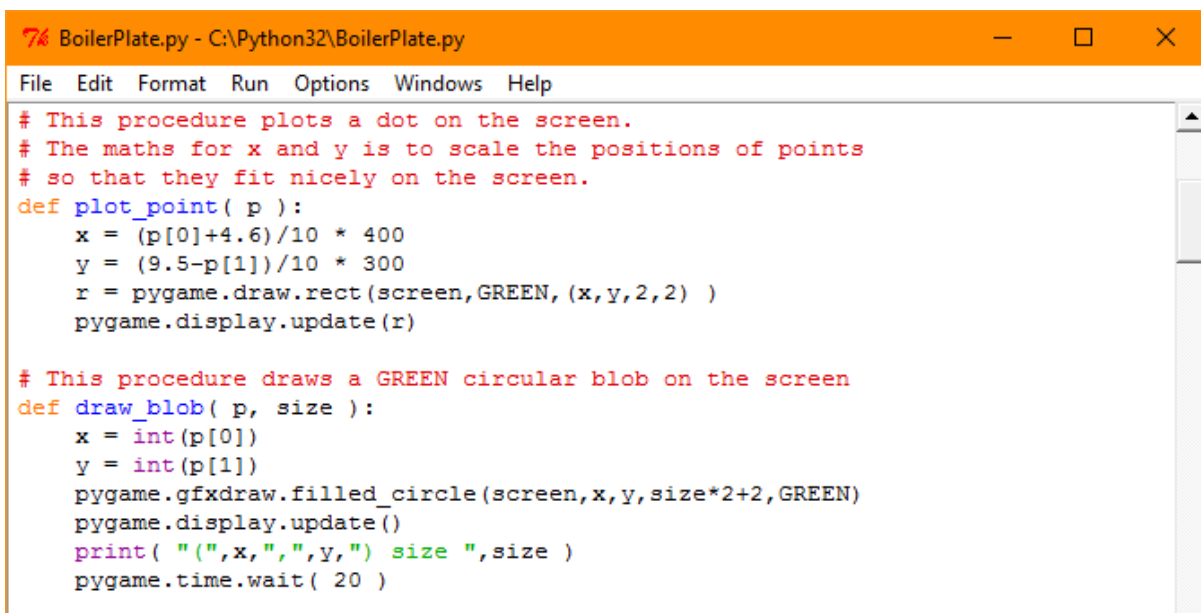
would print:

```
'mango'
```

However, using such a misleading name creates confusion for a programmer trying to read the code.

Example: Making Complex Code More Readable

1. Have a look at Python program shown below. It is a complex piece of code, used here to illustrate some of the techniques discussed to enhance the readability of code.
2. See if you can identify the following:
 - a) Comments to explain the program
 - b) Named blocks of code



```

76 BoilerPlate.py - C:\Python32\BoilerPlate.py
File Edit Format Run Options Windows Help
# This procedure plots a dot on the screen.
# The maths for x and y is to scale the positions of points
# so that they fit nicely on the screen.
def plot_point( p ):
    x = (p[0]+4.6)/10 * 400
    y = (9.5-p[1])/10 * 300
    r = pygame.draw.rect(screen, GREEN, (x, y, 2, 2) )
    pygame.display.update(r)

# This procedure draws a GREEN circular blob on the screen
def draw_blob( p, size ):
    x = int(p[0])
    y = int(p[1])
    pygame.gfxdraw.filled_circle(screen, x, y, size*2+2, GREEN)
    pygame.display.update()
    print( "(" , x , "," , y , ")" size " , size )
    pygame.time.wait( 20 )

```

3. Review the code you have created so far in ECDL Computing and see where you might improve its readability by using the techniques from this lesson.

9.5 REVIEW EXERCISE

1. A 'comment' in a computer program is:
 - a. Anything contained in quotes
 - b. Anything in the code that is before a procedure
 - c. The ASCII character '#'
 - d. Text to help document the program.

2. You should use a comment:
 - a. After every single line in a program.
 - b. Before every single line in a program.
 - c. To make it easier for someone else to understand your program.
 - d. Only for the most difficult function in a program.

3. Variable names should be:
 - a. Short to make the program as fast as possible.
 - b. At least eight letters long, so that other people won't be able to guess them.
 - c. Made from letters and numbers, not just letters.
 - d. Chosen to make your program more easily understood.

4. Python blocks of code are indicated by:
 - a. Surrounding them with quotes.
 - b. Starting them with '{' and ending them with '}'.
 - c. Starting them with '(' and ending them with ')'.
d. Indenting the block of code.

LESSON 10 – CONDITIONAL STATEMENTS

After completing this lesson, you should be able to:

- Define the programming construct term conditional statement. Outline the purpose of conditional statements in a program
- Define the programming construct term logic test. Outline the purpose of a logic test in a program
- Use Boolean logic expressions in a program
- Use IF...THEN...ELSE conditional statements in a program

10.1 SEQUENCE AND STATEMENTS



Concepts

In early computer languages each line of code of a program was one instruction and was complete in itself. Each instruction was followed in order by the computer, one after the other. This is called **sequential programming**.

In **sequential programming** instructions are executed, in order, one after another.

In computer languages a **statement** is the smallest standalone unit of code that is complete in itself.

An example of a statement is the `print()` statement:

```
print( 'Something to print')
```

When programmers use the term 'statement' they are referring to small components of code like `print()`.

10.2 IF STATEMENT



Concepts

In a flowchart, a decision box can be considered a complete standalone unit. The decision box has a True or False outcome that allows a choice to be made on what the code executes next.

A **Conditional Statement** is used to evaluate an expression as True or False. The outcome, a True or False value, determines what is done next.

In Python, conditional statements use a structured format and layout, and are written with the keyword **if** followed by a boolean expression to evaluate the condition, and then a colon.

For example:

```
Age = input ("How old are you?")
```

```
if (Age > 14 ) :
```

```
    print ( "You are younger than me" )
```

The indented block of code on the next line (the `print()` command in this case) will be obeyed if the expression is evaluated as true.

The boolean expression (12 (Age) > 14) in the if statement is called a **logic test**.

A **Logic test** is an expression that gives a Yes or No, True or False answer. The answer affects what code runs next.

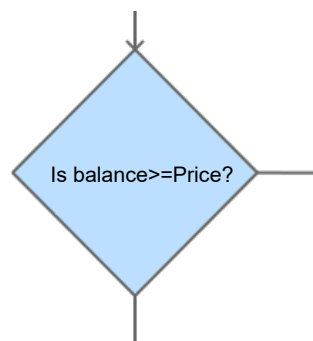
This is the structured format to use:

```
If expression
    statement if true
```

In pseudocode the logic test may be expressed in natural language such as:

‘Do I have enough Money?’

The same logic test would be shown as text in a flowchart like this:

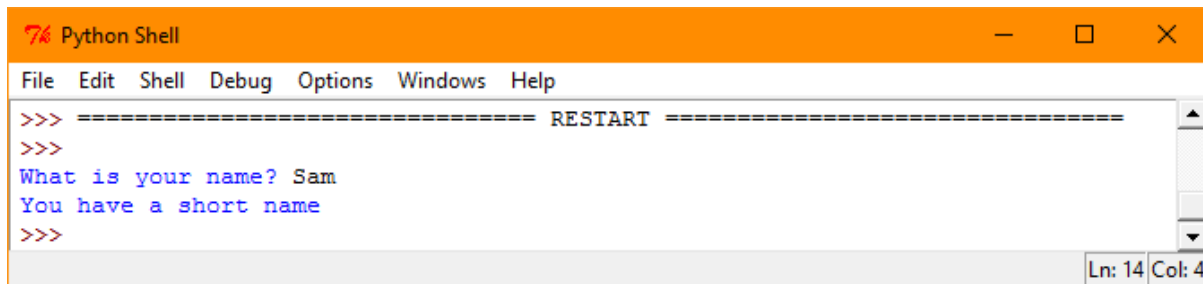


Example: A Logic Test

1. Open Python.
2. Create a new program file called NameLen.py Write the code as shown below. Be sure to use a colon as shown at the end of the line that starts with the keyword if.

```
NameLen.py - C:/Python32/NameLen.py
File Edit Format Run Options Windows Help
Name = input("What is your name? ")
if len( Name ) < 4 ):
    print( "You have a short name" )
|
Ln: 4 Col: 0
```

3. Run the program.
4. Respond to the prompt by typing either a short or a long name (more or fewer than 4 letters).
5. Observe the result.



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
What is your name? Sam
You have a short name
>>>
Ln: 14 Col: 4
```

10.3 IF...ELSE STATEMENT

Concepts

Python can allow for one block of indented code, a subroutine, to run if the logic test is True, and different block of code to run if the logic test is False.

The code is written with the if statement as before, and a new keyword **else**, followed by a colon, to add code that runs if the logic test is False.

This is the structure format to use:

```

if expression:
    statement if true
else:
    statement if false

```

For example:

```

Age = input ("How old are you?")
if (Age > 14 ):
    print ( "You are younger than me" )
else
    print ( "You are older than me" )

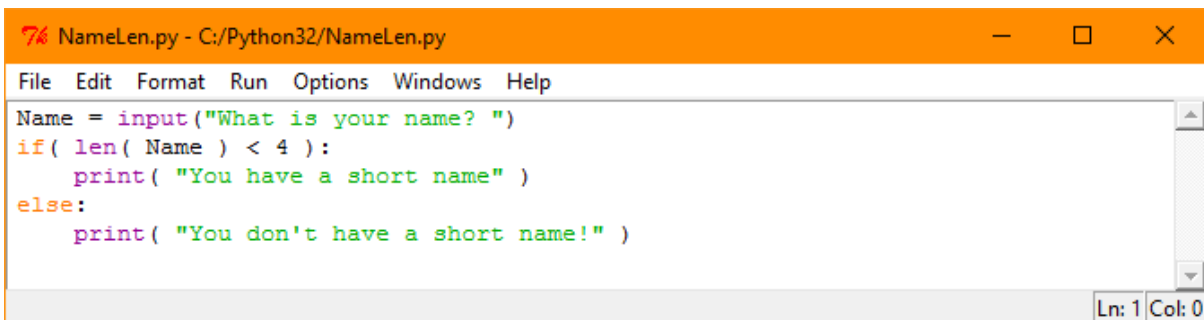
```

The indented block of code on the next line (the print() command in this case) will be obeyed if the expression is evaluated as true.

Note: The 'else' is not indented, it starts at the same indentation level as the 'if' keyword. When you try to the keyword 'else', Python will initially indent the word, so you will have to press the backspace key to get the correct level of indentation.

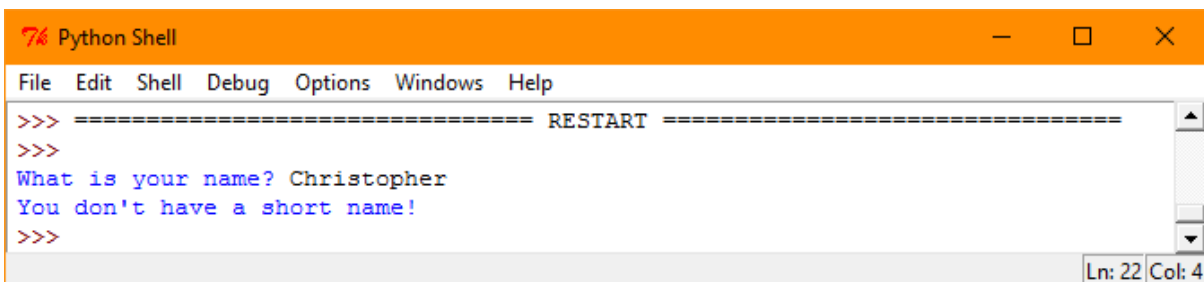
Example: An if..else Statement.

1. Open the NameLen.py example.
2. Modify it so that it reads as shown.
3. When you reach the point of typing in the 'else' press the backspace key to cancel one level of indentation.
4. Make sure there is a colon after the keyword else.



```
76 NameLen.py - C:/Python32/NameLen.py
File Edit Format Run Options Windows Help
Name = input("What is your name? ")
if( len( Name ) < 4 ):
    print( "You have a short name" )
else:
    print( "You don't have a short name!" )
Ln: 1 Col: 0
```

5. Run the code.
6. Type in a short name.
7. Run the code again.
8. Type in a longer name to see the code running for the case where the logic test is false.



```
76 Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
What is your name? Christopher
You don't have a short name!
>>>
Ln: 22 Col: 4
```

10.4 REVIEW EXERCISE

1. A Python conditional statement:

- a. Has the keyword 'if' in it
- b. Has the keyword 'def' in it
- c. Is a special kind of comment saying what conditions the code should be used in.
- d. Is a special kind of procedure that may in some conditions be a function instead.

2. In the following code:

```
if password == 'Voldemort' :  
    print( 'I guessed your password correctly' )  
else :  
    print( 'I did not guess your password correctly' )
```

...which of the following statements are correct?

- a. Both the first and second indented block will always be run.
- b. Neither the first nor the second indented block will ever be run
- c. There is a logic error because you cannot use == with a string.
- d. The first or second indented block may be run. Which one runs depends on the value of password.

LESSON 11 – PROCEDURES AND FUNCTIONS

After completing this lesson, you should be able to:

- Understand the term parameter. Outline the purpose of parameters in a program
- Understand the term procedure. Outline the purpose of a procedure in a program
- Write and name a procedure in a program
- Understand the term function. Outline the purpose of a function in a program
- Write and name a function in a program

11.1 SUBROUTINES



Concepts

One piece of code may be used more than once in different places in a program. This is called a subroutine.

A **subroutine** is a block of code that can be called up and run many times in a program. For example a subroutine could be used to display the date or time on the computer screen.

In Python the keyword 'def', (short for 'define'), is used to give a name to a subroutine. We can **call** or **invoke** a subroutine from anywhere in the entire program by referencing its name. The subroutine will run and when it has finished, the main program resumes from before the start of the subroutine.

A subroutine can perform a calculation, and return the answer or value to the program. This is called **returning** a value.

There are two types of subroutine.

Function

A Function is a subroutine that calculates a value for the program that contains it.

Procedure

A Procedure is a subroutine that executes an action, but does not return a value.

11.2 FUNCTIONS AND PROCEDURES



Concepts

Functions and procedures are made more flexible by having **parameters**.

A **parameter** is a special variable which is used in a subroutine to influence what it does.

Example: Calling Built-in Functions and Procedures

Python comes with some subroutines, i.e. functions and procedures, already defined.

input() and print() are both subroutines:

- input() is an example of a function. When called from within a program, it performs an action, and returns a value back to the code that called it. The

program resumes from where the input() function was called from in the program.

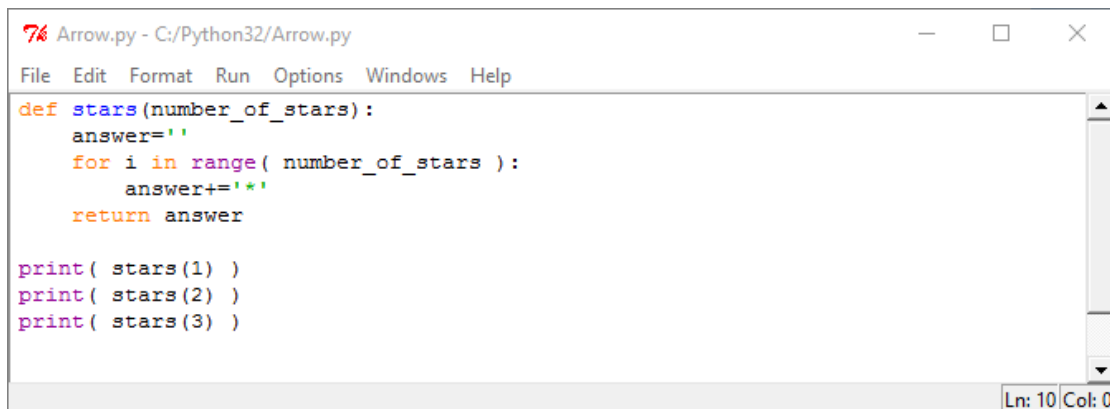
- print() is an example of a procedure. When called from within a program, it performs an action and afterwards the program resumes from where the print() procedure was called in.

Example: Creating a Function

This piece of code defines a function called 'stars' which has a **parameter** called 'number_of_stars' which determines the number of stars to return in a string.

There are three print statements in the program for us to check that the function is working correctly.

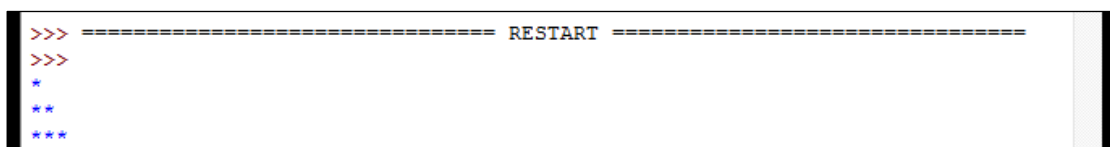
1. Open **Python**.
2. Create a new file called Arrow.py.
3. Type in the first block of code from the screenshot below, with the def statement and all the indented code lines to create a new function.
4. When you type in the three print statements, you call up the function three times.



```
Arrow.py - C:/Python32/Arrow.py
File Edit Format Run Options Windows Help
def stars(number_of_stars):
    answer=''
    for i in range( number_of_stars ):
        answer+='*'
    return answer

print( stars(1) )
print( stars(2) )
print( stars(3) )
Ln: 10 Col: 0
```

5. Run the program to see the result.



```
>>> ===== RESTART =====
>>>
*
**
***
```

Note: The program runs in sequence from the first print statement.

The first print statement prints one star on the first line.

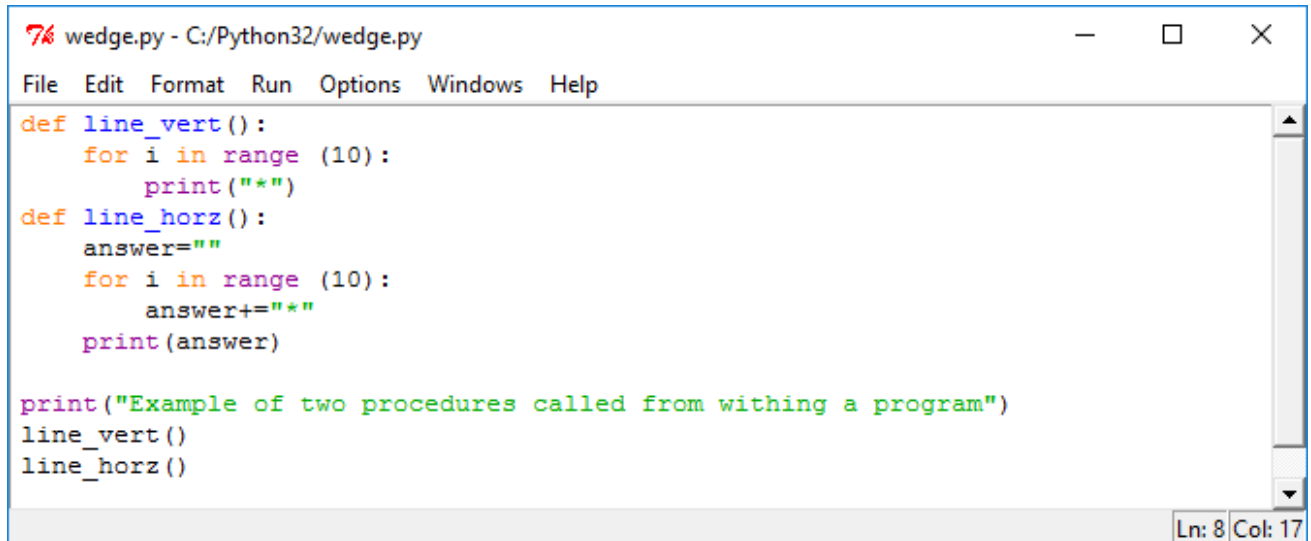
The second print statement prints two stars on the second line.

The third print statement prints three stars on the third line.

The quantity of stars printed on each line is dependent on the value of the parameter number_of_stars in each print statement.

Example: Creating Procedures

1. Open Python.
2. Create a new file called Lines.py.
3. Type in the two blocks of code from the screenshot below, with the two def statements and all the indented code lines to create two procedures.
4. Then type in the print statement and next, the two lines to invoke the two procedures in the main body of the program.
5. Save the program and run it to see the result.

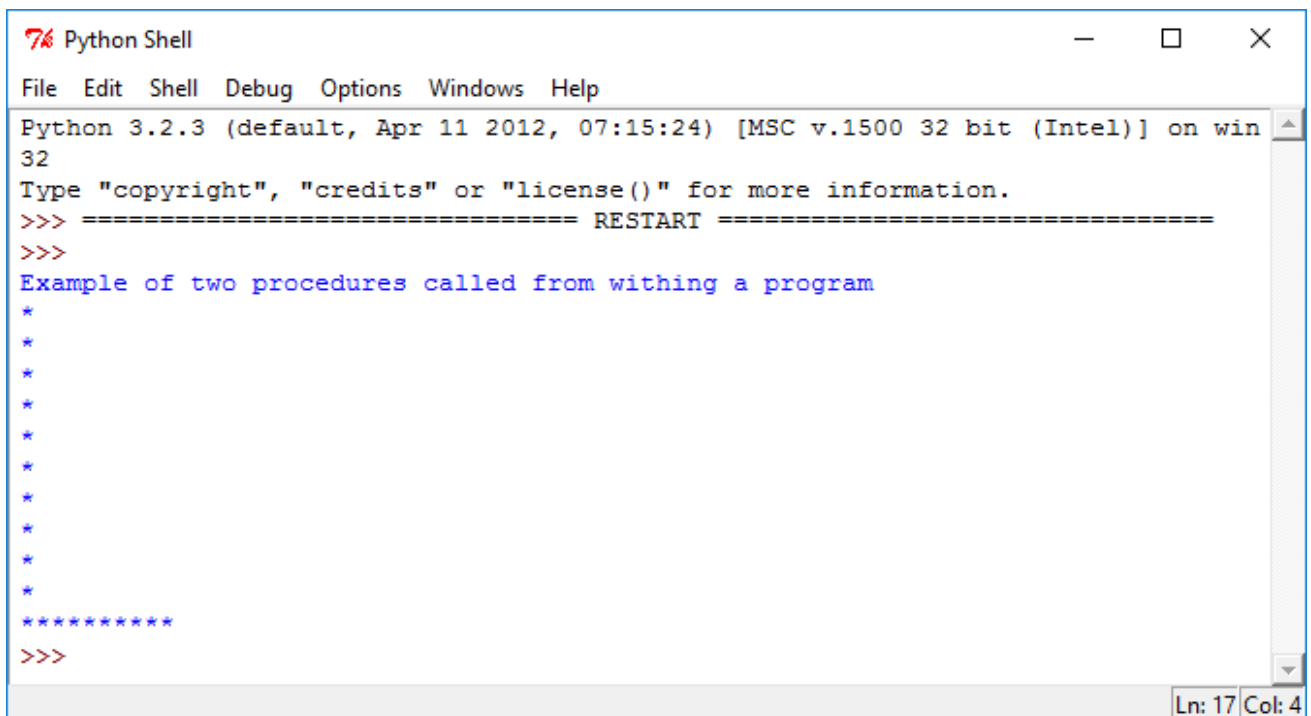


```

wedge.py - C:/Python32/wedge.py
File Edit Format Run Options Windows Help
def line_vert():
    for i in range (10):
        print("**")
def line_horz():
    answer=""
    for i in range (10):
        answer+="*"
    print(answer)

print("Example of two procedures called from withing a program")
line_vert()
line_horz()
Ln: 8 Col: 17

```



```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Example of two procedures called from withing a program
*
*
*
*
*
*
*
*
*
*
*****
>>>
Ln: 17 Col: 4

```

How the program works:

- The program runs in sequence starting with the print statement

- The program calls the procedure `line_vert` which prints a vertical line of stars on the screen. After executing this procedure, the program resumes with the next instruction which calls another procedure called `line_horz`.
- Both procedures run but don't return any values to the program.

11.3 REVIEW EXERCISE

1. A typical function in Python:
 - a. Starts with the keyword `def` and provides some code that returns a value
 - b. Starts with the keyword `abc` and provides some code that returns a value
 - c. Is a named piece of code that does a number of things and then stops the program.
 - d. Is a named piece of code that does a number of things and then restarts the Python shell.

2. A typical procedure in Python:
 - a. Consists of pseudocode which isn't meant for the computer to run.
 - b. Consists entirely of comments specifying what the program is intended to do.
 - c. Starts with a `#`
 - d. Is like a function, but does not return a value.

3. A parameter is:
 - a. A value which is passed in to a procedure or function for it to use
 - b. A way of measuring distance approximately, that can makes some code simpler.
 - c. A way for a program to do several things at the same time.
 - d. A block of code with a specified number of lines to it.

LESSON 12 – LOOPS

After completing this lesson, you should be able to:

- Define the programming construct term loop. Outline the purpose and benefit of looping in a program
- Recognise types of loops used for iteration: for, while, repeat
- Understand the term infinite loop
- Use iteration (looping) in a program like: for, while, repeat

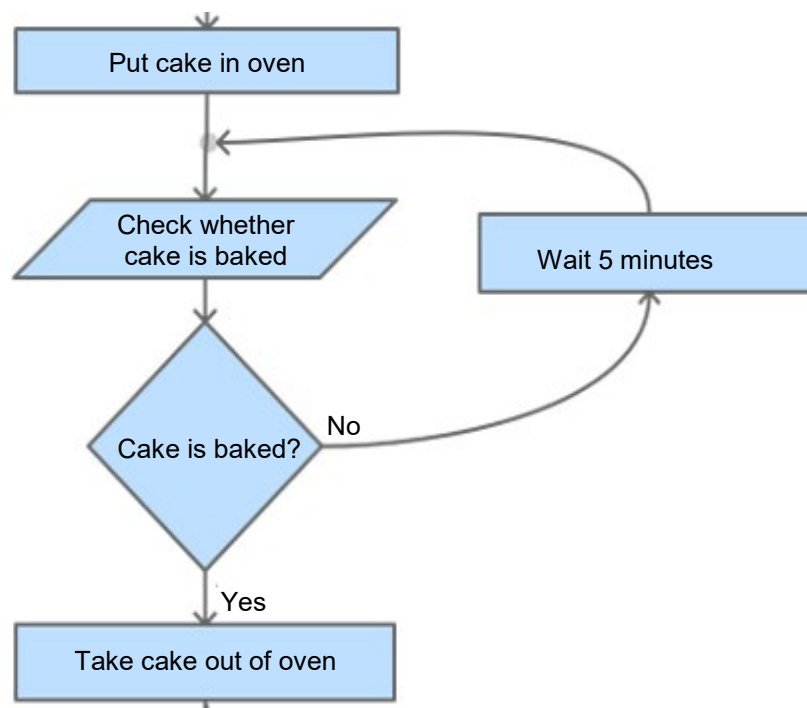
12.1 LOOPING

Concepts

Up to this point we have created programs that work sequentially, i.e. statements are executed in the order they are written, like following a recipe step by step. If we need to carry out the same step several times, we can use what's known as a **loop** in computing.

A **loop** is a piece of code that is run repeatedly under certain conditions.

A loop is represented in flowcharts where an arrow from a decision box goes back to an earlier box in the sequence. Examine the flowchart for baking the cake below. Checking the cake to see if it is baked is a step that could be repeated several times before the cake is ready to be taken out of the oven. This repeatable step is represented by a loop – check cake, if not baked, wait 5 minutes then check again.



A loop in a flowchart

Using a loop to repeat an action is one of the most useful techniques in coding. Looping code is present in most programming languages with slight variations on naming conventions e.g. 'repeat' is used to determine how many times to loop in some languages; we'll see later in this lesson that 'while' does the same thing in Python.

Python uses the **'for'** statement to make loops.

The **for** statement has several parts to it.

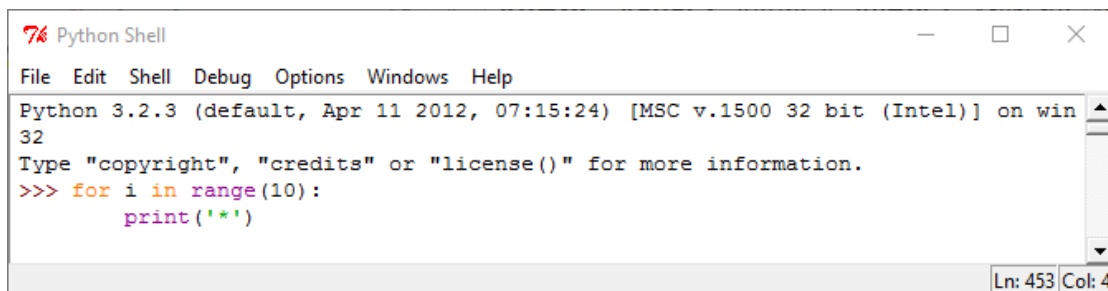
```
for i in range( 10 ):
    statement
```

- The keyword 'for' is followed by the name of a variable, in this case i, keeping count of the repeats of the loop. The variable i will have an initial value 0, and after each time the loop is run, it increments to 1 then 2, and so on up to 9.
- The words 'in range()' with a number inside the parentheses followed by a colon, indicating how many times the loop will run
- An indented line of code with a statement will be executed 10 times.

Example: Attempting to Print a Row of Ten Stars

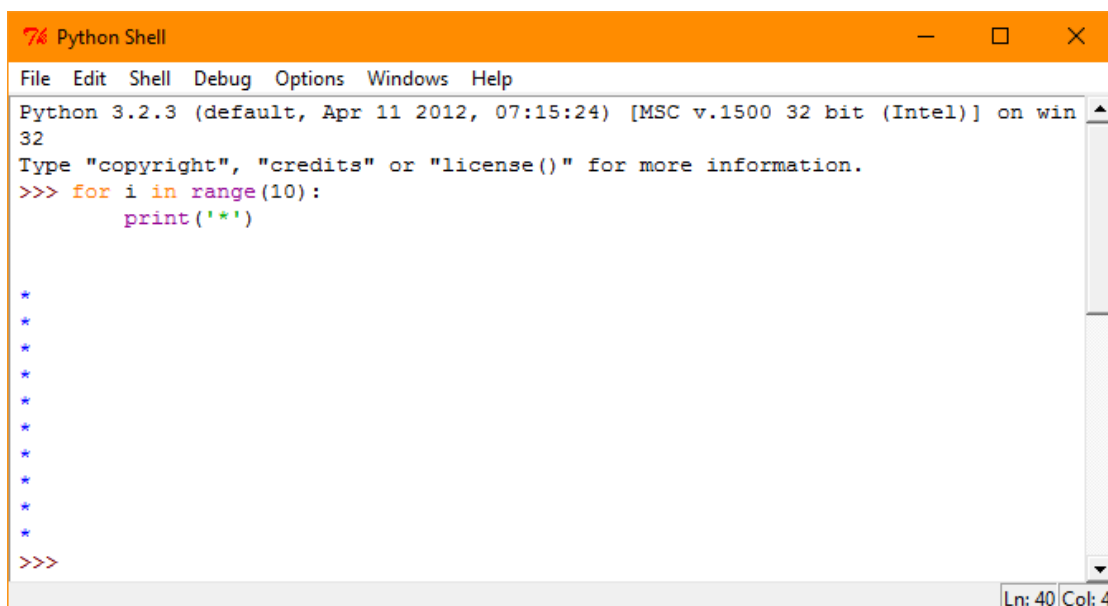
This example shows an attempt to print a row of ten stars.

1. Open Python.
2. Type in the following code.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> for i in range(10):
    print('*')
```

3. Press Enter to run the code.



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> for i in range(10):
    print('*')
*
*
*
*
*
*
*
*
*
*
>>>
```

12.2 LOOPING WITH VARIABLES



Concepts

In the previous example each star was displayed on a line of its own. We wanted a row of stars.

One different approach would be:

```
print('*****')
```

This will work but it is not efficient – what if you wanted to print 599 stars?

This is where loops save a lot of programming time and lead to shorter code. We can write a loop to print 10 stars, 42 stars or 599 stars, depending on a value.

How the program works:

- Print a row of 10 stars by using a loop is to build up the row of stars in a variable. Then, once the row is ready, the code can print it. This approach will use the + operator to add the string '*' onto a variable called answer.

```
answer = answer + '*'
```

This line of code uses the current value of answer, adds a star onto the end, and puts the result back into answer.

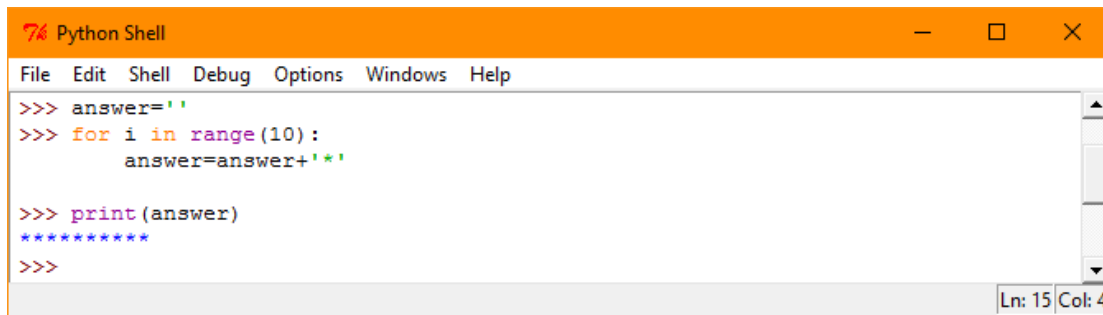
- The variable named answer must first be initialised before it is used. To initialise it to a string with three stars use:

```
answer = '***'
```

To initialise answer to a string that is empty, remove the three stars from the line above. Keep the two quotation marks! Without them it's an error. The two quotation marks at the start and end tell Python that answer is a string.

Example: Row of Stars using a Variable

1. Open **Python**.
2. Initialise the variable called answer to be an empty string.
3. Use a loop to add 10 stars onto the string.
4. Print the string.



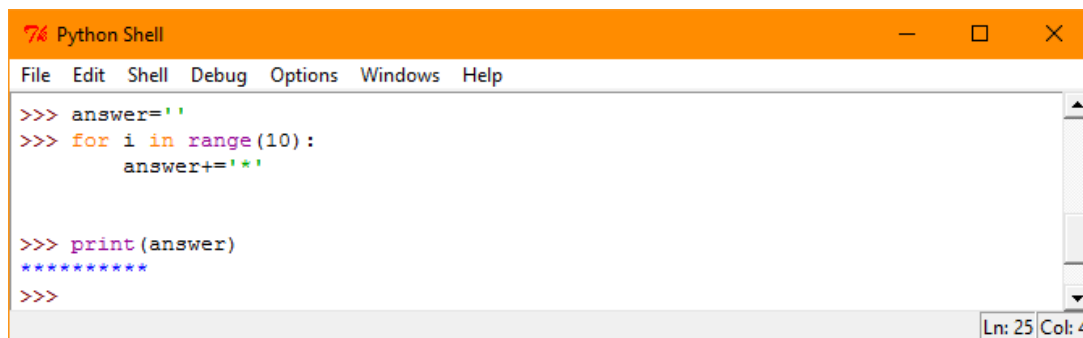
```

Python Shell
File Edit Shell Debug Options Windows Help
>>> answer=''
>>> for i in range(10):
>>>     answer=answer+'*'
>>>
>>> print(answer)
*****
>>>
Ln: 15 Col: 4

```

There is a shorthand way of writing `answer=answer+'*'` which uses a new operator `+=`.

5. Write the same code using the `+=` operator, as shown below:



```

Python Shell
File Edit Shell Debug Options Windows Help
>>> answer=''
>>> for i in range(10):
>>>     answer+='*'
>>>
>>> print(answer)
*****
>>>
Ln: 25 Col: 4

```

12.3 VARIATIONS ON LOOPS

Concepts

We have seen that loops have a particular structure and layout in Python.

```

for i in range( 10 ):
    statement

```

The indented code block after the 'for' line is said to be 'inside the loop'.

- Code before the loop is executed and then the loop is executed.
- Code inside the loop is executed multiple times.
- Code immediately after the loop is executed only after the loop has finished repeating the code in the indented block.

So far we have looked at the basic loop but there are other variations of loops that we can use as well.

Another kind of loop uses a logic test rather than `range()`. It's called a while loop.

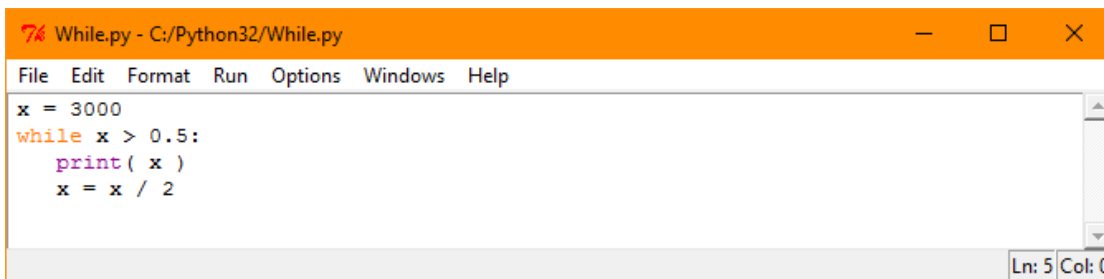
A **while loop** tests a boolean expression and repeats execution of the loop as long as (ie While) the boolean expression is true.

How the program works:

- The example that follows will initialise a variable x to 3000, and keep dividing by 2 while x is greater than 0.5.
- Then it will stop.

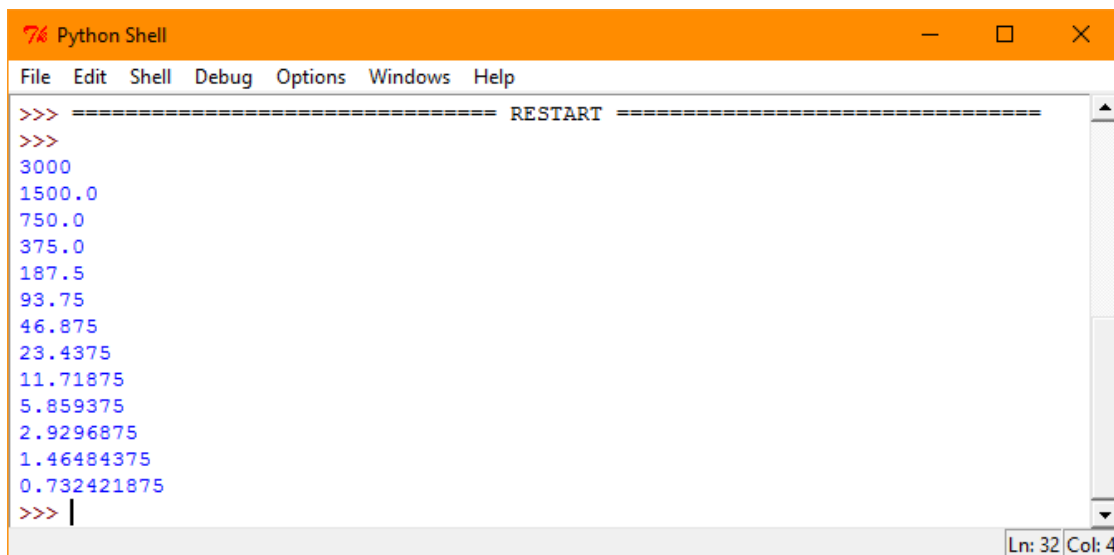
Example: A While Loop

1. Open Python.
2. Create a new program file called While.py.
3. Type in the code as shown below:



```
File Edit Format Run Options Windows Help
x = 3000
while x > 0.5:
    print( x )
    x = x / 2
Ln: 5 Col: 0
```

4. Run the program.



```
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
3000
1500.0
750.0
375.0
187.5
93.75
46.875
23.4375
11.71875
5.859375
2.9296875
1.46484375
0.732421875
>>> |
Ln: 32 Col: 4
```

In a while loop, if the logic test is always true, the loop will repeat forever. It then becomes an infinite loop.

An infinite loop never stops. It uses the key words "while true", followed by a colon.

As we usually want a program to stop at some point, infinite loops usually indicate an error.

Example: An Infinite Loop

1. Open the program While.py.
2. Modify the logic test so that the line reads While True:

```

x = 3000
while True:
    print( x )
    x = x / 2
    
```

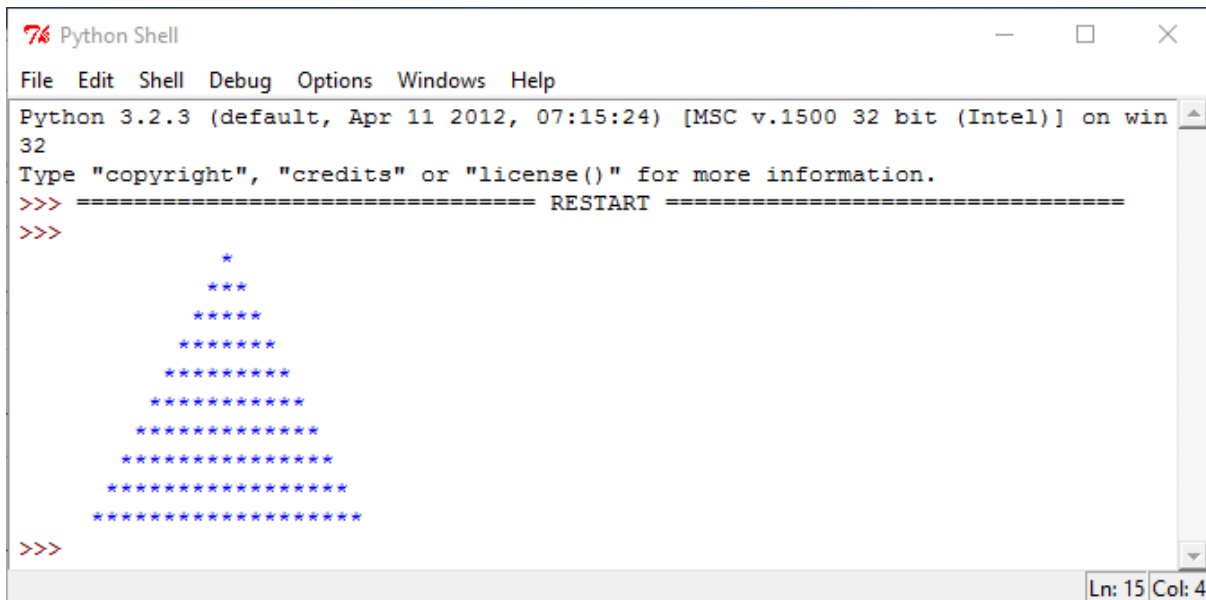
3. Run the program.
4. You can stop the program by opening the menu and clicking on Shell, Restart Shell.

12.4 PUTTING IT ALL TOGETHER

Concepts

Example: Drawing an Arrowhead

Python can be used to write a program that draws an arrowhead like the one shown below:



```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
          *
         ***
        *****
       *******
      *********
     **********
    ***********
   *************
  ***************
 *****************
 *****************
 *****************
 >>>
Ln: 15 Col: 4

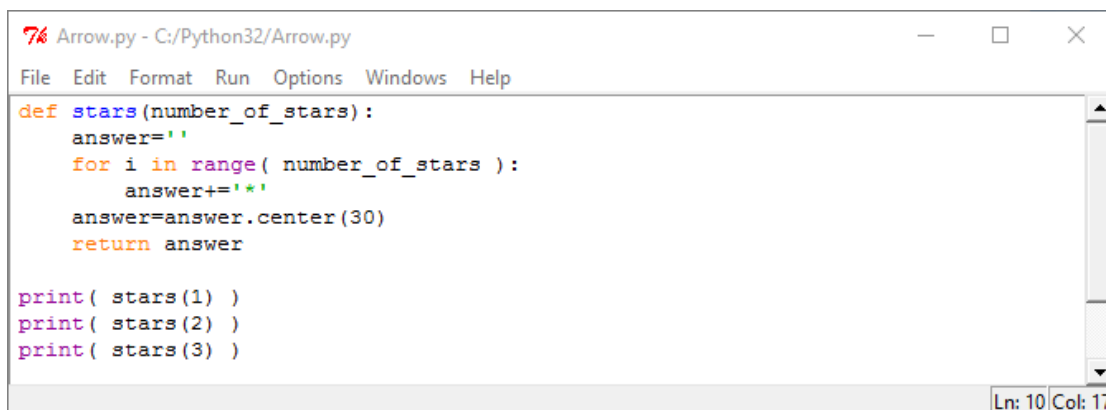
```

This task could be done with lots of `print()` statements. However, it can be done with fewer instructions by using loops, functions and procedures.

To draw the arrow, each row of stars needs to be horizontally centered, rather than aligned to the left. There is a function to do that. It's called `center()`. If the variable named `answer` holds a string of stars, the following line of code adds spaces before and after the stars to make 30 characters in all.

```
answer=answer.center(30)
```

1. Open the `Arrow.py` Example from your previous lesson.
2. Modify the code as follows so that it will centre the stars.



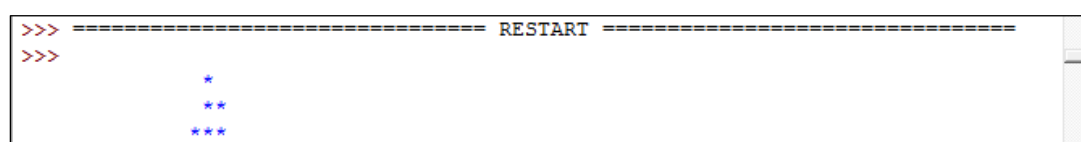
```

Arrow.py - C:/Python32/Arrow.py
File Edit Format Run Options Windows Help
def stars(number_of_stars):
    answer=''
    for i in range( number_of_stars ):
        answer+='*'
    answer=answer.center(30)
    return answer

print( stars(1) )
print( stars(2) )
print( stars(3) )
Ln: 10 Col: 17

```

3. Run the code to get the following result.



```

>>> ===== RESTART =====
>>>
          *
         **
        ***

```


- To get a more arrow-like shape the first row should have one star, the second three stars and the third row five stars. Change the parameter value in each print statement, as follows, to fix that:

```

7% Arrow.py - C:/Python32/Arrow.py
File Edit Format Run Options Windows Help
def stars(number_of_stars):
    answer=''
    for i in range( number_of_stars ):
        answer+='*'
    answer=answer.center(30)
    return answer

print( stars(1) )
print( stars(3) )
print( stars(5) )
Ln: 10 Col: 17
    
```

- Run the code to get the following result:

```

>>> ===== RESTART =====
>>>
          *
         ***
        *****
    
```

We can create an expression and write a subroutine to create the arrow head and reduce the amount of code needed. We need 10 rows of stars to make the arrowhead.

Consider the expression:

$$i*2+1$$

- When i=0 the value of the expression is 1
- When i=1 the value of the expression is 3
- When i=2 the value of the expression is 5
- When i=3 the value of the expression is 7
- When i=4 the value of the expression is 9
- When i=5 the value of the expression is 11
- When i=6 the value of the expression is 13
- When i=7 the value of the expression is 15
- When i=8 the value of the expression is 17
- When i=9 the value of the expression is 19

1. Give the new subroutine the name `arrow_head()`.
2. Invoke `arrow_head()` as the last line of the program.

```

7% Arrow.py - C:/Python32/Arrow.py
File Edit Format Run Options Windows Help
def stars(number_of_stars):
    answer=''
    for i in range( number_of_stars ):
        answer+='*'
    answer=answer.center(30)
    return answer

def arrow_head():
    for i in range(10):
        print(stars(i*2+1))

arrow_head()
Ln: 8 Col: 17

```

Note: The `arrow_head()` subroutine is a **procedure**, because it does not return a value.

The `stars()` subroutine is a **function**, because it returns a value.

1. Take a moment to review how the program works before running the code:
 - The program begins by executing the procedure `arrow_head()`.
 - The variable `i` will have ten values from (0-9) and so the `arrow_head()` procedure will run ten times, each time calling the function `stars()`. This is an example of one subroutine calling another.
 - The function `stars()` contains a variable called `answer` this variable will hold a string of stars for each print statement.
 - Subroutines calling other subroutines is one of the most important ways that large programs are broken up into smaller pieces, each piece doing some small well defined part of the overall task.
2. Run `Arrow.py` to see the arrow head that the code draws.

```

>>> ----- RESTART -----
>>>
          *
         ***
        *****
       *******
      *********
     **********
    ***********
   *************
  ****************
 *****************
 *****************
  *****************
   *****************
    *****************
     *****************
      *****************
       *****************
        *****************
         *****************
          *****************

```

12.5 REVIEW EXERCISE

1. What is the main purpose of a loop?
 - a. To stop the computer running off the end of a program.
 - b. To go back to the beginning of a program.
 - c. To connect one computer to another.
 - d. To do some instructions repeatedly.

2. A loop that goes on forever is called?
 - a. A pixel loop.
 - b. A broken loop.
 - c. An infinite loop.
 - d. An 'iffy-loop'.

3. To count through several items, we typically use:
 - a. a loop the loop
 - b. random numbers
 - c. a while loop
 - d. a for loop

4. To repeat some action several times without counting we typically use:
 - a. a loop the loop
 - b. random numbers
 - c. a while loop
 - d. a for loop

LESSON 13 – LIBRARIES

After completing this lesson, you should be able to:

- Understand the term event. Outline the purpose of an event in a program
- Use event handlers like: mouse click, keyboard input, button click, timer
- Use available generic libraries like: math, random, time

13.1 USING LIBRARIES



Concepts

Functions and procedures provide a way of reusing code. Two ways of reusing code on a larger scale are:

Libraries

A library is a collection of already written procedures and functions that can be reused from a program. A Library saves the work of writing procedures and functions from scratch.

- The source code of library functions is not visible in the program.
- A library is added by giving the name of the library in an 'import' command near the start of a Python program.

ECDL Computing learning materials uses these libraries:

Random, Math and Time

Standard libraries that come with Python and that provide functions for working with numbers

pygame

An optional library for working with graphics and animation.

Boilerplate code

Is code that is used with little or no modification in a project, for example boilerplate code may provide functions to show images on multiple pages of a document and this code is included as it will be called many times in a program.

- Boilerplate code is source code that is visible in the program.
- Boilerplate code is used by including all its source code in the program being worked on.

When using boilerplate code or a library it is not necessary to know the details of how the algorithms in boilerplate code or library work.

The learning materials include boilerplate code to initialise and use the libraries.

13.2 STANDARD LIBRARIES

Concepts

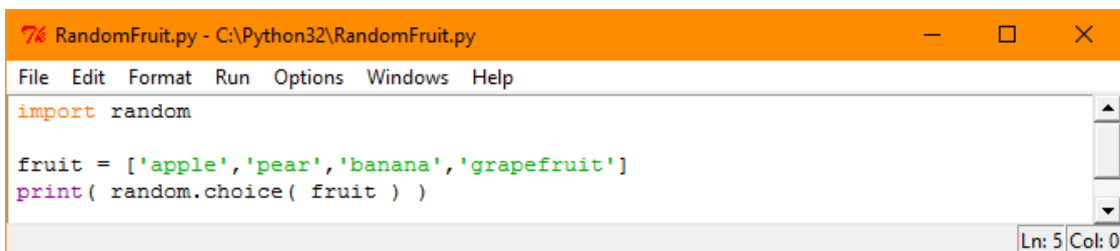
The Import Statement

Python uses the import statement to make code in libraries available to use.

There are two versions of the import statement:

The first version uses the keyword **import** followed by the name of the library.

When using this version of import, functions from the library have the name of the library first, then a dot and then the function name. An example is shown below:




```
7% RandomFruit.py - C:\Python32\RandomFruit.py
File Edit Format Run Options Windows Help
import random

fruit = ['apple', 'pear', 'banana', 'grapefruit']
print( random.choice( fruit ) )
Ln: 5 Col: 0
```

Example of import statement without naming functions

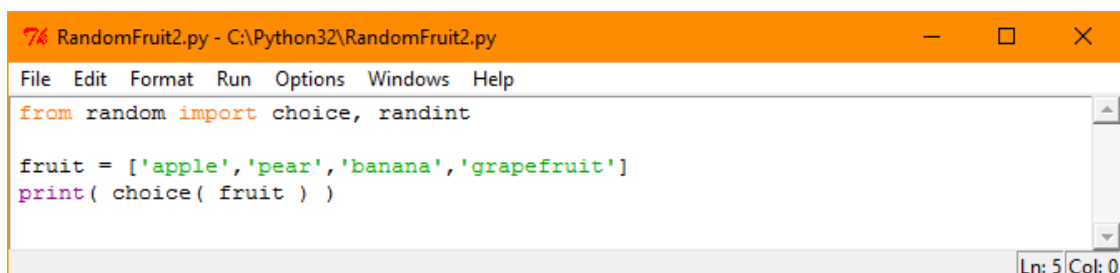
Running this example will give a random fruit name, for example, as shown below:



```
7% Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
apple
Ln: 16 Col: 4
```

An alternative version of the import statement names each function to import. It uses the keyword **from**, then the name of the library, then the keyword **import** then the names of functions to import with a comma between them.

When using this version of import, functions from the library do not need to have the library name before them. An example is shown below:



```
7% RandomFruit2.py - C:\Python32\RandomFruit2.py
File Edit Format Run Options Windows Help
from random import choice, randint

fruit = ['apple', 'pear', 'banana', 'grapefruit']
print( choice( fruit ) )
Ln: 5 Col: 0
```

Example of import statement with named functions

Three Standard Libraries

Three standard libraries that come with Python are:

LIBRARY	PURPOSE
time	Functions related to time, such as for printing today's date, time, or day of the week.
random	A library for generating random numbers
math	A library with mathematical functions,

time Library

There are two particularly useful functions in the time library. They are usually used together:

FUNCTION	PURPOSE
strftime	A function that takes a 'time object' and converts it to a string. This is called formatting. There are many options for exactly how this function formats the time.
gmtime	A function that gives a 'time object' for the current time of day.

The strftime function uses 'format specifiers' to describe exactly how it wants the string result to look. For example, the letters '%Y' are replaced by the year, so the string 'In the year %Y' could be changed to the string 'In the year 2001' by strftime.

Here is a table of the format specifiers for strftime:

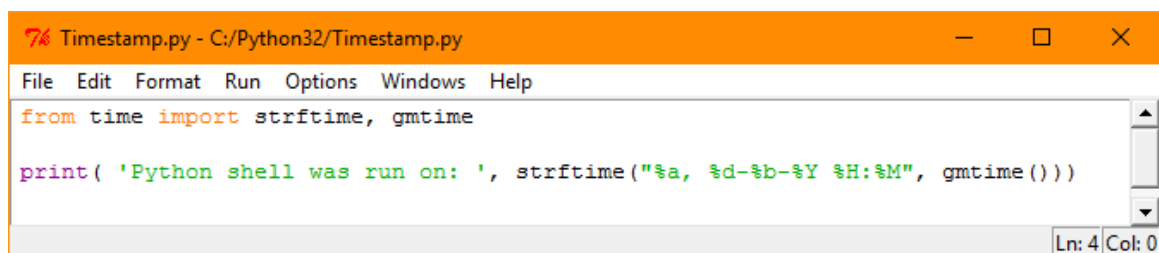
FORMAT	MEANING	EXAMPLE
%a	Day of week (short)	Mon
%A	Day of week (full)	Monday
%b	Month name (short)	Sep
%B	Month name (long)	September
%c	Date and Time	15/09/76 17:22:56
%d	Day of month	15
%H	Hour (24 hour clock)	17

%l	Hour (12 hour clock)	5
%j	Day of the year	350
%m	Month as a number	9
%M	Minute as a number	22
%p	Either AM or PM	PM
%S	Seconds as a number	56
%u, %w or %W	Week number	15
%x	Date	15/09/76
%X	Time	17:22:56
%y	Year (two digits)	76
%Y	Year (four digits)	1976
%Z	Time Zone	GMT Standard Time

Example: A timestamp

In this example strftime is used to print a timestamp. This example uses many of the format options from the table above.

1. Create a new program called Timestamp.py.
2. Write the code as shown below.



```

76 Timestamp.py - C:/Python32/Timestamp.py
File Edit Format Run Options Windows Help
from time import strftime, gmtime

print( 'Python shell was run on: ', strftime("%a, %d-%b-%Y %H:%M", gmtime()))
Ln: 4 Col: 0

```

3. Run the program to get the following result. The time and date will be the current time and date, rather than the time and date shown here.



```

76 Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
Python shell was run on: Sun, 09-Sep-2001 01:46
>>>
Ln: 41 Col: 4

```


random Library

Two useful functions in the random library are choice and randint.

FUNCTION	PURPOSE
choice	A function that picks one random item from a list
randint	A function that picks a random integer between two integers. randint(1,6), for example, gives a random number between 1 and 6, like the roll of a dice.

Example: Rolling a dice

In this example, randint is used to roll a dice 10 times.

1. Create a new program called RollDice.py.
2. Write the code as shown below.

```

74 RollDice.py - C:/Python32/RollDice.py
File Edit Format Run Options Windows Help
from random import randint

print("Rolling a dice 10 times...")
for i in range(0,10):
    print( randint( 1,6 ) )
Ln: 6 Col: 0
    
```

3. Run the program.

```

74 Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
Rolling a dice 10 times...
6
2
5
3
6
2
1
4
6
4
>>>
Ln: 54 Col: 4
    
```

math Library

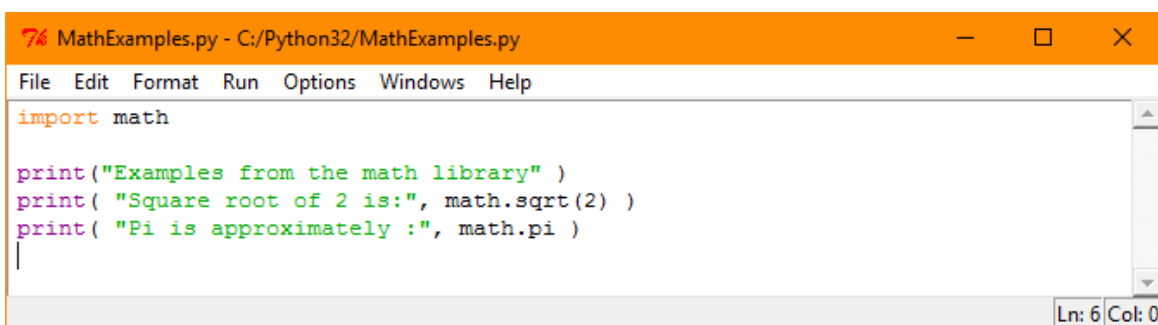
Some of the functions in the math library are:

FUNCTION	PURPOSE
log10	Logarithm base 10. $\log_{10}(100000)$ is 5. It counts the number of zeroes in that case.
pow	Raise to the power. $\text{pow}(10,3)$ multiplies 10 by itself three times, so gives 1000 as the result.
sqrt	Square root. $\text{sqrt}(16)$ is 4. $\text{sqrt}(25)$ is 5.
sin	Sine of an angle measured in radians.
cos	Cosine of an angle measured in radians.
tan	Tangent of an angle measured in radians.
factorial	Factorial of a number. $\text{factorial}(5)$ is $5 * 4 * 3 * 2 * 1$ which is 120

In addition the math library has the value for the mathematical constant pi.

Example: Using the math Library

1. Create a new program called MathExample.py.
2. Write the code as shown below.



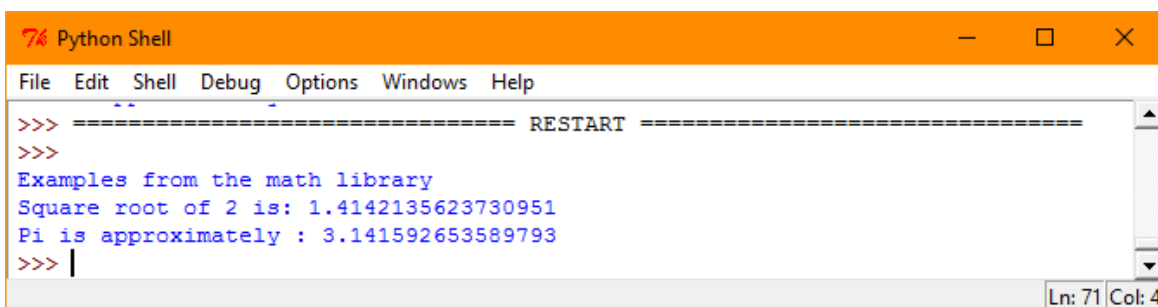
```

74 MathExamples.py - C:/Python32/MathExamples.py
File Edit Format Run Options Windows Help
import math

print("Examples from the math library" )
print( "Square root of 2 is:", math.sqrt(2) )
print( "Pi is approximately :", math.pi )
|
Ln: 6 Col: 0

```

3. Run the program



```

74 Python Shell
File Edit Shell Debug Options Windows Help
>>> ===== RESTART =====
>>>
Examples from the math library
Square root of 2 is: 1.4142135623730951
Pi is approximately : 3.141592653589793
>>> |
Ln: 71 Col: 4

```

13.3 EVENTS



Concepts

The flow of a program can depend on certain actions happening. Users can trigger these actions by, for example, using the mouse to click on a graphic they see on screen, pressing a key on the keyboard or clicking a link in a browser window. In Python programming these actions are called events.

Events are used extensively in games. Events that determine the movement of game's character are triggered by user input, for example: using the arrow keys to move the character in a particular direction, using a mouse click to make the character jump etc.

Some events may be specific to hardware. In an alarm system which uses a computer to monitor for intruders, a person moving near a sensor or pressure on a pressure mat could cause or trigger an event.

In some computers, the battery being low would trigger an event. This may cause a message to display to suggest recharging the battery.

Event Handlers

An Event handler is a piece of code designed to do something once an event has been triggered. For example the up, down, left, right arrows on a keyboard can control the movement of a player in a game. Depending on which event occurs (Up, down, left, right), the program will execute the appropriate 'event handler' code which will carry out the correct movement of the player.

Some computer languages have a system of 'registering' functions that are called when certain events occur. We can recognise these functions by their names which begin with the word 'On' followed by the name of the event. For example 'OnClick' is called when a user generates an event by clicking anywhere on the screen.

Example: Exploring Event Handlers

Python has a library called turtle. It is a simple drawing package included in the Python installation. To use this library we must use the import command at the start of our program.

```

76 Turtle_drawing.py - C:/Python32/Turtle_drawing.py
File Edit Format Run Options Windows Help
import turtle #import the turtle library
screenr = turtle.Screen() #give the screen a reference
pointer = turtle.Turtle() #give the turtle a reference

def h1():
    pointer.sety(pointer.ycor()+10) #move the pointer 10 pixels up
def h2():
    pointer.sety(pointer.ycor()-10) #move the pointer 10 pixels down
def h3():
    pointer.forward(10) #move the pointer 10 pixels to the right
def h4():
    pointer.back(10) #move the pointer 10 pixels to the left
def h5():
    pointer.penup() # Stop drawing
def h6():
    pointer.pendown() #start drawing
def h7():
    pointer.pencolor("black") #change the drawing colour to black
def h8():
    pointer.pencolor("white") #change the drawing colour to white.
def h9(x,y):
    pointer.left(180) #change the pointer direction by 180 degrees.
def h10():
    pointer.color("red") #change pointer fill colour

screenr.onkey(h1, "Up") #when the up arrow is pressed run h1 subroutine
screenr.onkey(h2, "Down") #when the down arrow is pressed run h2 subroutine
screenr.onkey(h3, "Right") #when the right arrow is pressed run h3 subroutine
screenr.onkey(h4, "Left") #when the left arrow is pressed run h4 subroutine
screenr.onkey(h5, "p") #when the letter p is pressed run h5 subroutine
screenr.onkey(h6, "d") #when the letter d is pressed run h6 subroutine
screenr.onkey(h7, "b") #when the letter b is pressed run h7 subroutine
screenr.onkey(h8, "w") #when the letter b is pressed run h8 subroutine
screenr.onclick(h9) #when mouse is clicked on the screen run h9 subroutine
screenr.ontimer(h10,5000) #when timer times out run h10 subroutine
screenr.listen() # check to see if any event has happened

turtle.mainloop() #keep looping
Ln: 1 Col: 0

```

How the program works:

The comments in the program explain what each of the ten subroutines do. The program continuously loops, checking on each pass to see if an event has happened. When an event is triggered, the program calls the subroutine assigned to that event. When it finishes it returns to the main body of the program.

- The first four onkey event handlers are used to determine if any arrow keys have been pressed on your keyboard. Subroutines h1-h4 are invoked once a keypress is detected. These are keyboard events.

- The next four onkey event handlers are used to detect if either p,d,b or w keys are pressed on the keyboard, again calling the associated subroutines h5-h8. These are keyboard events.
- The next event handler is used to detect a mouse click and triggers and event if detected and then runs the h9 subroutine. This is a click event.
- The last event handler is used to trigger and event when a timer times out. Once the time has elapsed, the subroutine h10 is run.

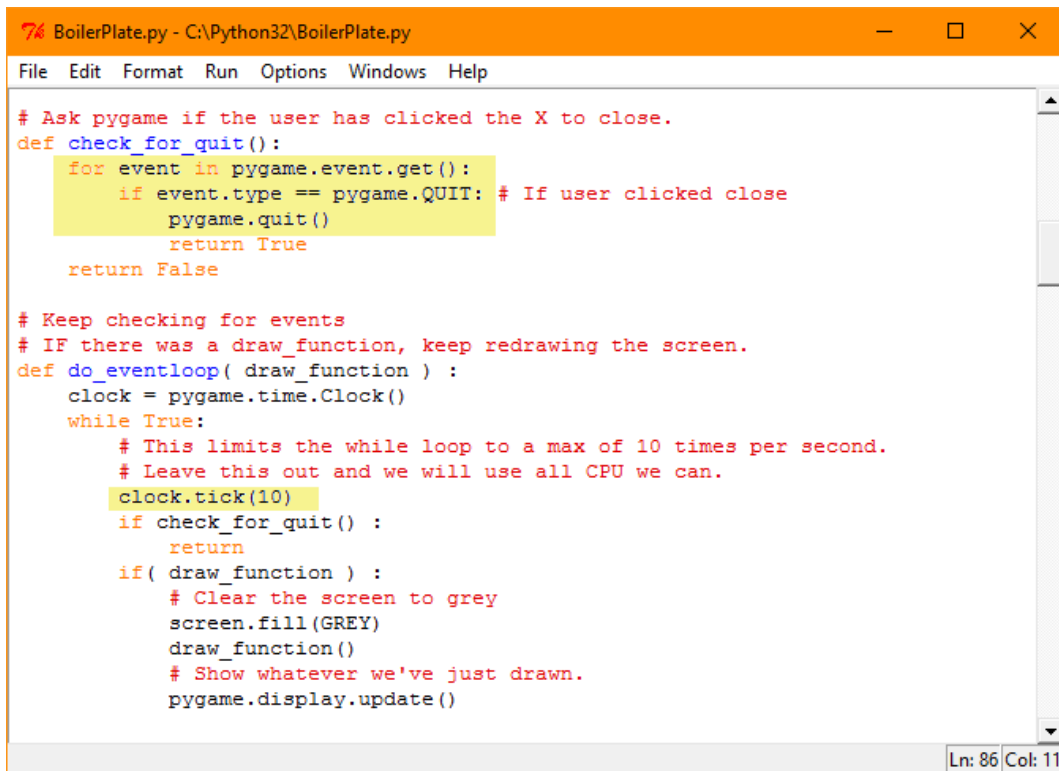
Note: the “screenr.listen” command is used to check for any events.

- Type in the code in the screenshot above, without the comments.
- Save the program as Turtle_drawing.py.
- Run the program.
- Note how the pointer changes from black to red after 5 seconds. This is the timer event triggering in the program.
- Note how when you press the arrow keys on the keyboard the pointer moves and draws lines. These are keyboard events being triggered for each arrow press.
- Press p on the keyboard turns the pen off, another keyboard event.
- Press d on the keyboard turns the pen on, another keyboard event.
- Pressing b or w on the keyboard changes the pen ink colour. These are keyboard events.
- Finally click your mouse anywhere on the screen. The mouse pointer turns 180 degrees. You have just triggered a click event in the program.

13.4 PYGAME LIBRARY

Concepts

In Python, and with Pygame, code for handling events is written to repeatedly request information to check whether any event has happened. The following screenshots are from the moving grass example later in this lesson. For now, let's take a first look at two common events.

A screenshot of a Python IDE window titled 'BoilerPlate.py - C:\Python32\BoilerPlate.py'. The window has a menu bar with 'File', 'Edit', 'Format', 'Run', 'Options', 'Windows', and 'Help'. The code editor shows the following Python code with two sections highlighted in yellow: the 'for' loop in the 'check_for_quit()' function and the 'clock.tick(10)' call in the 'do_eventloop()' function. The status bar at the bottom right shows 'Ln: 86 Col: 11'.

```
74 BoilerPlate.py - C:\Python32\BoilerPlate.py
File Edit Format Run Options Windows Help

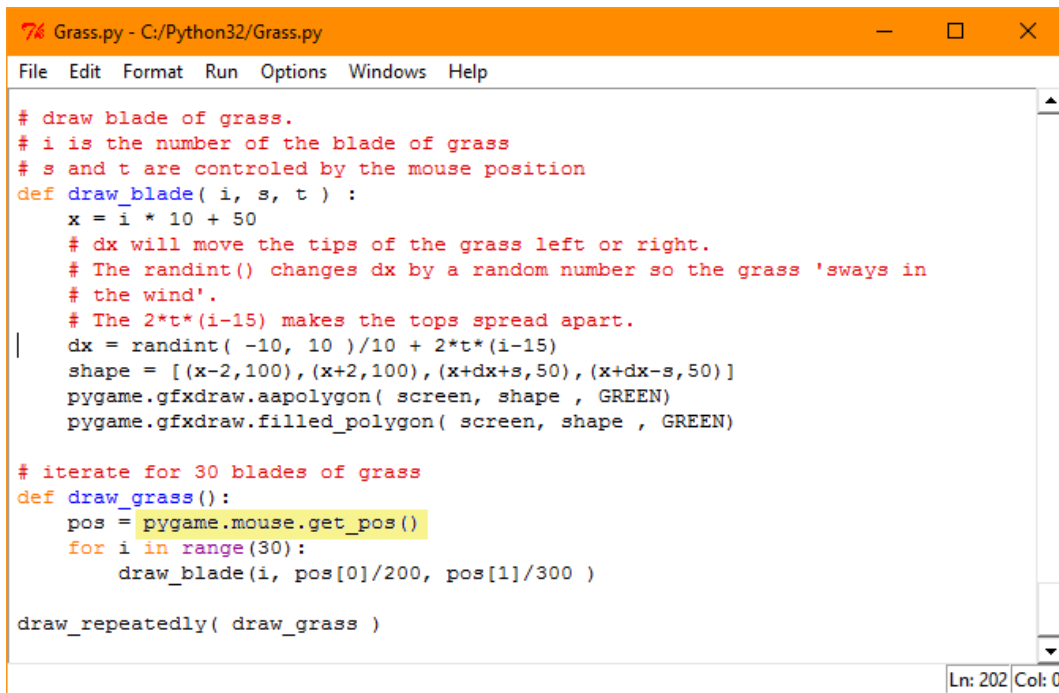
# Ask pygame if the user has clicked the X to close.
def check_for_quit():
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # If user clicked close
            pygame.quit()
            return True
    return False

# Keep checking for events
# IF there was a draw_function, keep redrawing the screen.
def do_eventloop( draw_function ) :
    clock = pygame.time.Clock()
    while True:
        # This limits the while loop to a max of 10 times per second.
        # Leave this out and we will use all CPU we can.
        clock.tick(10)
        if check_for_quit() :
            return
        if( draw_function ) :
            # Clear the screen to grey
            screen.fill(GREY)
            draw_function()
            # Show whatever we've just drawn.
            pygame.display.update()
```

Event handling highlighted in BoilerPlate.py

In the `check_for_quit()` function, a for loop repeatedly checks to see if the Quit event has occurred in Pygame. The QUIT event is triggered when the user clicks on the X in the top right of the Pygame window. If that event is detected, the `check_for_quit()` function returns True. If it is not detected, `check_for_quit()` returns False.

The second piece of highlighted code, the call to `clock.tick()` function is related to event handling too; it sets how frequently the while loop runs which in turn determines how often the screen is updated.



```

Grass.py - C:/Python32/Grass.py
File Edit Format Run Options Windows Help

# draw blade of grass.
# i is the number of the blade of grass
# s and t are controlled by the mouse position
def draw_blade( i, s, t ) :
    x = i * 10 + 50
    # dx will move the tips of the grass left or right.
    # The randint() changes dx by a random number so the grass 'sways in
    # the wind'.
    # The 2*t*(i-15) makes the tops spread apart.
    dx = randint( -10, 10 )/10 + 2*t*(i-15)
    shape = [(x-2,100), (x+2,100), (x+dx+s,50), (x+dx-s,50)]
    pygame.gfxdraw.aapolygon( screen, shape , GREEN)
    pygame.gfxdraw.filled_polygon( screen, shape , GREEN)

# iterate for 30 blades of grass
def draw_grass():
    pos = pygame.mouse.get_pos()
    for i in range(30):
        draw_blade(i, pos[0]/200, pos[1]/300 )

draw_repeatedly( draw_grass )
Ln: 202 Col: 0

```

Event handling highlighted in Grass.py

Another event example highlighted above, is in the `draw_grass()` procedure. The `mouse.get_pos()` is called to get the position of the mouse which is used in positioning the moving blades of grass.

13.5 BOILERPLATE CODE

Concepts

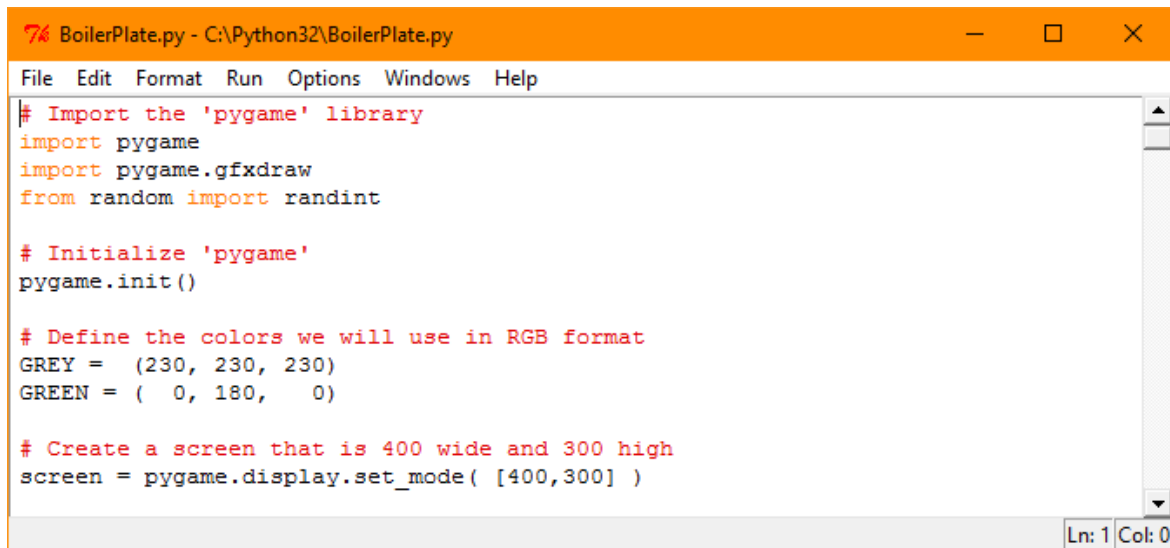
To use the Pygame library, a program needs to add the library, initialise it, define colours and do several other actions. Boilerplate code is provided with these learning materials to perform these actions.

We don't need to know how boilerplate code works in detail. We need to know:

- That tuples are used to define some colours used in painting the screen.
- That the boilerplate code imports from the random library to get the function `randint`.
- That it has an event loop, and what events are.
- That it uses `randint` to generates a random integer.

Example: Exploring BoilerPlate.py

1. Locate and open `BoilerPlate.py` supplied with these learning materials. (you don't need to type boilerplate.py code in yourself).



```
BoilerPlate.py - C:\Python32\BoilerPlate.py
File Edit Format Run Options Windows Help
# Import the 'pygame' library
import pygame
import pygame.gfxdraw
from random import randint

# Initialize 'pygame'
pygame.init()

# Define the colors we will use in RGB format
GREY = (230, 230, 230)
GREEN = ( 0, 180,  0)

# Create a screen that is 400 wide and 300 high
screen = pygame.display.set_mode( [400,300] )
Ln: 1 Col: 0
```

2. Look at the lines after the comment # Import the 'pygame' library. The three lines with 'import' are all about using libraries:

- 'import pygame' allows us access to the pygame library so that it is available to use.
- 'import pygame.gfxdraw' allows us access to an optional extension to pygame that draws shapes more smoothly.
- The 'random' library provides random numbers. The import command allows us access one function, randint, that generates random integers.

The remaining code:

- Initialises the Pygame library. That library has many variables inside it that need to have their initial values.
- It defines and initialises two colours, GREY and GREEN. The round parentheses indicate that these colours are defined as tuples with three integer values.
- It creates a variable called screen that is a window 400 pixels wide and 300 pixels high.

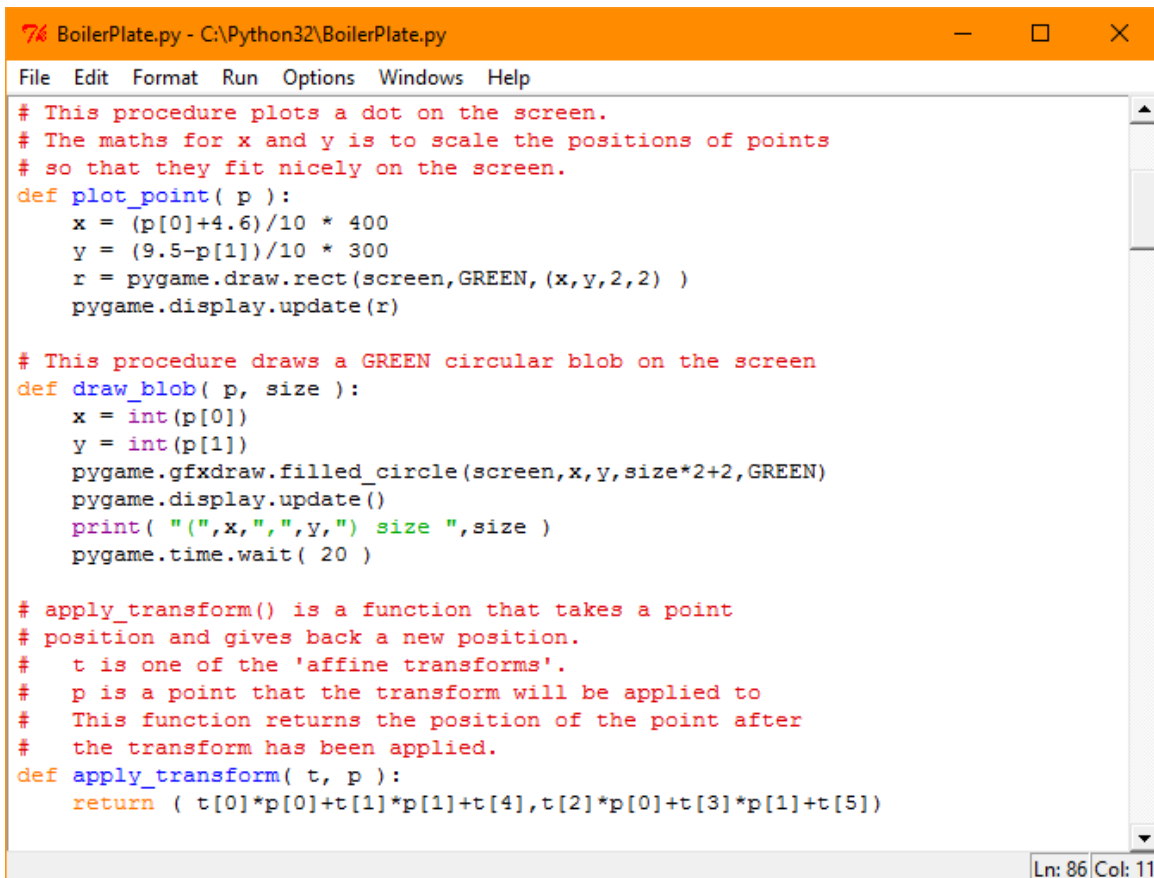
3. Scroll down - the boilerplate code defines some procedures for plotting points and blobs. Read the comments describing each subroutine

The subroutines are explained below in a bit more detail than they are in the comments. This section of code provides utility functions and procedures.

- plot_point will plot a green dot on the screen. It uses the numbers 400 and 300, which are the width and height of the screen, to scale the point to the right position on the screen.

- `draw_blob` will draw a green circular blob on the screen. It also prints the coordinates of the blob.
- `apply_transform` applies an 'affine transformation' to a point. This is used to distort shapes. It can make a shape bigger/smaller, rotate it, reflect it, shift it left or right, up or down. For this course we only need to know that the affine transform takes some coordinate and returns a different one as a result.

The screenshot that follows was given as an example of comments and code structure in an earlier section.



```

7% BoilerPlate.py - C:\Python32\BoilerPlate.py
File Edit Format Run Options Windows Help
# This procedure plots a dot on the screen.
# The maths for x and y is to scale the positions of points
# so that they fit nicely on the screen.
def plot_point( p ):
    x = (p[0]+4.6)/10 * 400
    y = (9.5-p[1])/10 * 300
    r = pygame.draw.rect(screen, GREEN, (x,y,2,2) )
    pygame.display.update( r )

# This procedure draws a GREEN circular blob on the screen
def draw_blob( p, size ):
    x = int(p[0])
    y = int(p[1])
    pygame.gfxdraw.filled_circle(screen,x,y,size*2+2, GREEN)
    pygame.display.update()
    print( "(" ,x, "," ,y, ")" size ",size )
    pygame.time.wait( 20 )

# apply_transform() is a function that takes a point
# position and gives back a new position.
# t is one of the 'affine transforms'.
# p is a point that the transform will be applied to
# This function returns the position of the point after
# the transform has been applied.
def apply_transform( t, p ):
    return ( t[0]*p[0]+t[1]*p[1]+t[4],t[2]*p[0]+t[3]*p[1]+t[5] )
Ln: 86 Col: 11

```

4. Scroll down - the next part of the boilerplate code is concerned with events.

Read the comments on the event processing code.

Notice that `check_for_quit()` is a function. It returns a boolean value.

Notice that `do_eventloop()` is a procedure. It does have the return keyword in it, but that return does not return a value.

The event processing code is:

- `check_for_quit()` checks an event. It asks Pygame if the user has clicked on the 'X' in the top right hand corner of the graphics window. If they have, it's time to quit that window.

- `do_eventloop()` has an infinite loop, the 'while True', that keeps checking for quit and redrawing the graphics screen. The repeated drawing of the graphics screen is relevant for animation.

`do_eventloop()` uses an unusual technique. The parameter that is passed in to `do_eventloop()` can be a function. If it actually is a function, then `do_eventloop` will call that function each time it draws the graphic screen.

```

BoilerPlate.py - C:\Python32\BoilerPlate.py
File Edit Format Run Options Windows Help

# Ask pygame if the user has clicked the X to close.
def check_for_quit():
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # If user clicked close
            pygame.quit()
            return True
    return False

# Keep checking for events
# IF there was a draw_function, keep redrawing the screen.
def do_eventloop( draw_function ) :
    clock = pygame.time.Clock()
    while True:
        # This limits the while loop to a max of 10 times per second.
        # Leave this out and we will use all CPU we can.
        clock.tick(10)
        if check_for_quit() :
            return
        if( draw_function ) :
            # Clear the screen to grey
            screen.fill(GREY)
            draw_function()
            # Show whatever we've just drawn.
            pygame.display.update()

```

Ln: 86 Col: 11

5. Scroll down

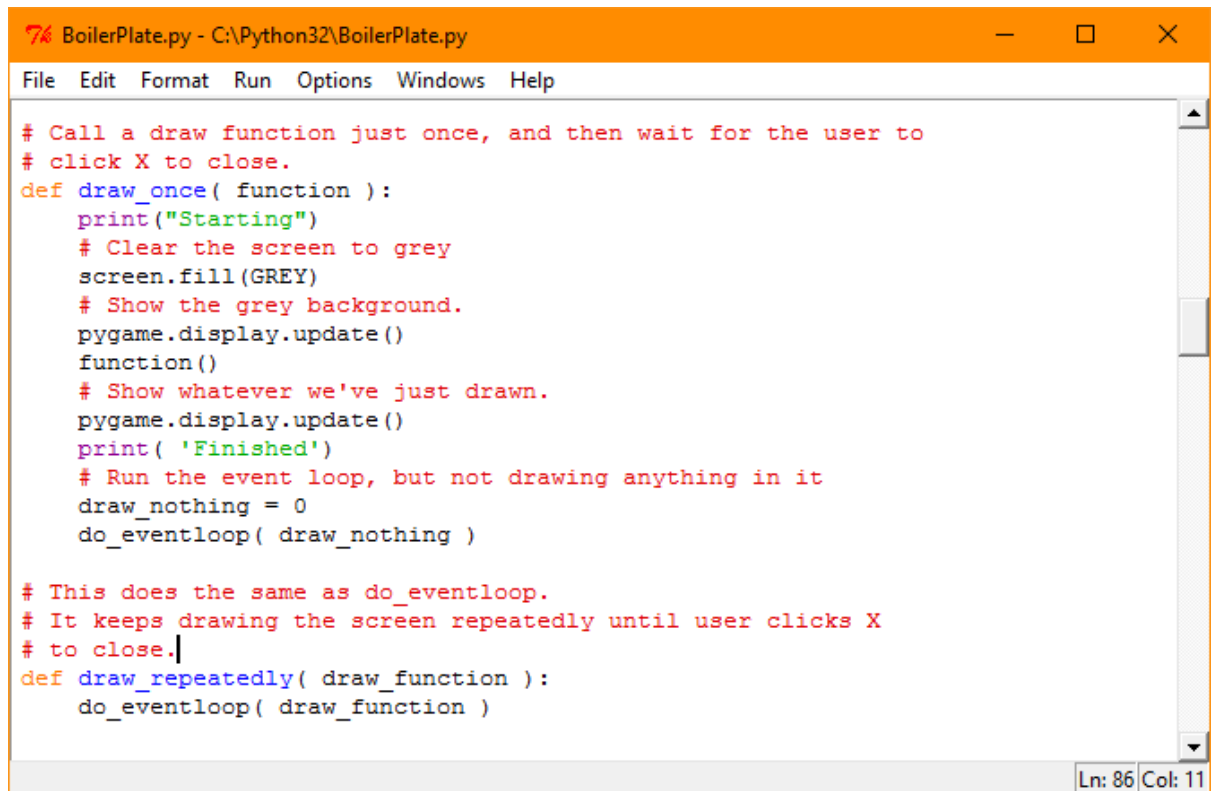
The remaining code gives two ways to draw the screen.

Read the comments for the procedures.

These two procedures give two choices for ways to use the event loop.

- `draw_once()` will run our drawing code once, then wait for the user to quit the graphics screen.
- `draw_repeatedly()` will run our drawing code over and over, checking each time to see if the user has quit. This procedure is suitable for animation.

The `draw_once()` function has a sequence of actions. These include printing out 'Starting' and 'Finished' and waiting for quit in the event loop.



```
76 BoilerPlate.py - C:\Python32\BoilerPlate.py
File Edit Format Run Options Windows Help

# Call a draw function just once, and then wait for the user to
# click X to close.
def draw_once( function ):
    print("Starting")
    # Clear the screen to grey
    screen.fill(GREY)
    # Show the grey background.
    pygame.display.update()
    function()
    # Show whatever we've just drawn.
    pygame.display.update()
    print( 'Finished' )
    # Run the event loop, but not drawing anything in it
    draw_nothing = 0
    do_eventloop( draw_nothing )

# This does the same as do_eventloop.
# It keeps drawing the screen repeatedly until user clicks X
# to close.
def draw_repeatedly( draw_function ):
    do_eventloop( draw_function )

Ln: 86 Col: 11
```

Example: Make a Copy of the BoilerPlate.py

To use Pygame, make a copy of the boilerplate code, and then add your code to it.

1. Open the BoilerPlate.py file.
2. Use steps from the example 'To Create and Save a Program'. Use from the SaveAs step onwards to save BoilerPlate.py with a new name you choose.
3. Check that you have made a copy by opening the new copy of the file.

13.6 DRAWING USING THE LIBRARIES

Concepts

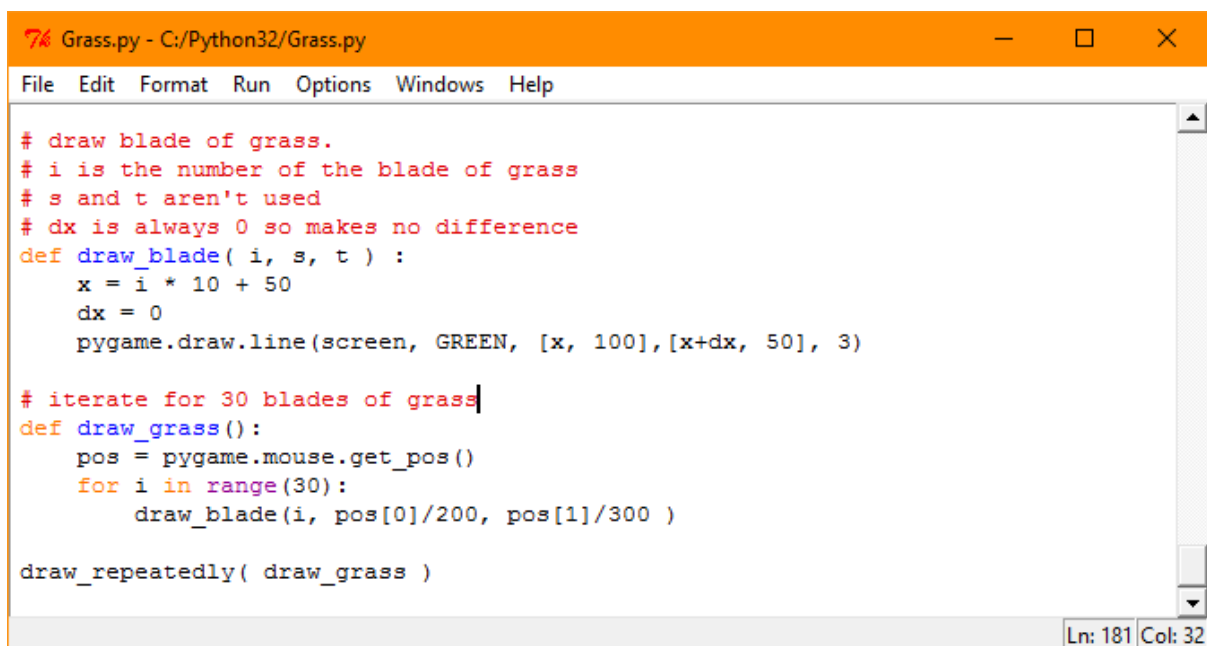
For professional 3D games, graphic artists create custom art for many of the features of the worlds and monsters in the game. However, a lot of detail can be generated from code. For example, artists do not draw every blade of grass and animate each blade moving in the breeze. Instead they create one or a few different blades of grass and then use procedures and loops to draw and animate many blades of grass moving.

Pygame has a 2D graphics library. Pygame code can use a loop and random numbers to show the principle of getting the computer to do the repetitive work. Code can draw and animate one blade of grass and that code can then be put into a loop to draw and animate many blades of grass.

In the following Pygame example the 'blades of grass' are just green lines.

Example: Blades of Grass

1. Make a copy of the boilerplate.py code,
2. Call the copy Grass.py
3. Add the following code on the end:



```

76 Grass.py - C:/Python32/Grass.py
File Edit Format Run Options Windows Help

# draw blade of grass.
# i is the number of the blade of grass
# s and t aren't used
# dx is always 0 so makes no difference
def draw_blade( i, s, t ) :
    x = i * 10 + 50
    dx = 0
    pygame.draw.line(screen, GREEN, [x, 100],[x+dx, 50], 3)

# iterate for 30 blades of grass
def draw_grass():
    pos = pygame.mouse.get_pos()
    for i in range(30):
        draw_blade(i, pos[0]/200, pos[1]/300 )

draw_repeatedly( draw_grass )

Ln: 181 Col: 32

```

4. Review the following points about this code. Check how these points relate to the actual code.

`draw_grass()` has a loop and calls `draw_blade()` 30 times.

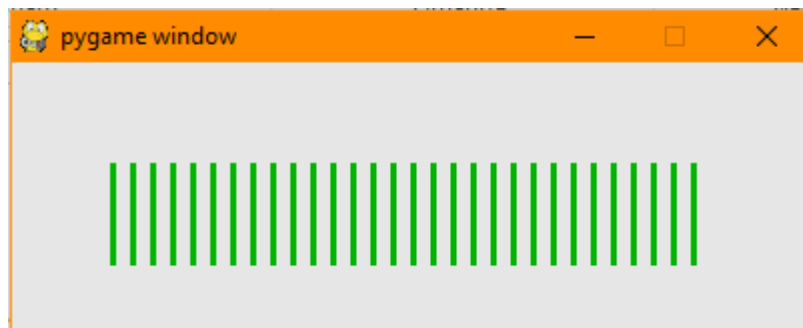
`draw_blade()` in turn calls `draw_line()` in the pygame library, and draws a GREEN line of width 3.

Changing GREEN to be the tuple (0,0,255) would give blue grass instead.

- `draw_blade()` has three parameters, `i`, `s` and `t`, but at this stage only one of the parameters, `i`, is used. Subroutines do not have to use all the parameters that are passed to them.
- `i` is the number of the blade of grass. `i` sets the position of the blades of grass, `x`.

- The 'x = i * 10 + 50' derives x from i. The formula takes the blade of grass number, which could be 0, 1, 2, 3... and changes it to an x coordinate, 50, 60, 70, 80... The *10 makes the blades of grass 10 pixels apart, and the +50 starts the sequence at x=50. Change these values slightly and run the program to see different spacing and start position.
- draw_grass() invokes the function mouse.get_pos().
- mouse.get_pos() gets the position of the mouse as a tuple. The tuple contains the x and y position of the mouse. The code divides x by 200 and y by 300 to get the values of the parameters s and t for draw_blade.

5. Run the program to see the blades of grass:



The picture of grass currently has straight lines for the grass. To make the blades of grass taper, replace the code that says draw_line().

6. Make the following changes to the code:

```

Grass.py - C:/Python32/Grass.py
File Edit Format Run Options Windows Help
|
# draw blade of grass.
# i is the number of the blade of grass
# s and t aren't used
# dx is always 0 so makes no difference
# Now the grass is a bit more pointy.
def draw_blade( i, s, t ) :
    x = i * 10 + 50
    dx = 0
    shape = [[x-2,100],[x+2,100],[x+dx+s,50],[x+dx-s,50]]
    pygame.gfxdraw.aapolygon( screen, shape , GREEN)
    pygame.gfxdraw.filled_polygon( screen, shape , GREEN)

# iterate for 30 blades of grass
def draw_grass():
    pos = pygame.mouse.get_pos()
    for i in range(30):
        draw_blade(i, pos[0]/200, pos[1]/300 )

draw_repeatedly( draw_grass )
Ln: 180 Col: 0

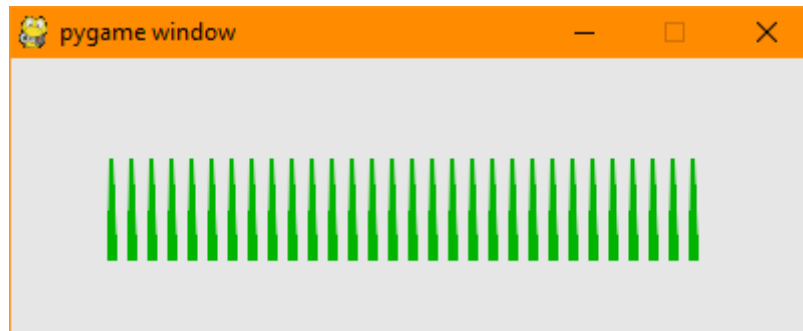
```

7. Relate the description below to the changes in the code.

The biggest change here is creating a new variable, shape.

- The new list variable 'shape' holds coordinates for four corners of that blade of grass.
- draw.line has been got rid of. The two gfxdraw commands draw the outline of the shape and then the inside, respectively. Change the outline to a different colour to GREEN to show this.

8. Run the code to see blades of grass that taper.



9. The next step makes two more changes to what the program does:

- Makes the grass spread out
- Makes the grass sway

There is only one significant code change involved. It affects the variable dx. dx moves the tips of the grass left or right. 0 means no displacement left or right. A negative dx moves the tip left. Positive moves right. The change in the code changes the instruction $dx = 0$ to have a calculation for dx.

10. Change the code as follows:

```

7% Grass.py - C:/Python32/Grass.py
File Edit Format Run Options Windows Help

# draw blade of grass.
# i is the number of the blade of grass
# s and t are controlled by the mouse position
def draw_blade( i, s, t ) :
    x = i * 10 + 50
    # dx will move the tips of the grass left or right.
    # The randint() changes dx by a random number so the grass 'sways in
    # the wind'.
    # The 2*t*(i-15) makes the tops spread apart.
    dx = randint( -10, 10 )/10 + 2*t*(i-15)
    shape = [(x-2,100), (x+2,100), (x+dx+s,50), (x+dx-s,50)]
    pygame.gfxdraw.aapolygon( screen, shape , GREEN)
    pygame.gfxdraw.filled_polygon( screen, shape , GREEN)

# iterate for 30 blades of grass
def draw_grass():
    pos = pygame.mouse.get_pos()
    for i in range(30):
        draw_blade(i, pos[0]/200, pos[1]/300 )

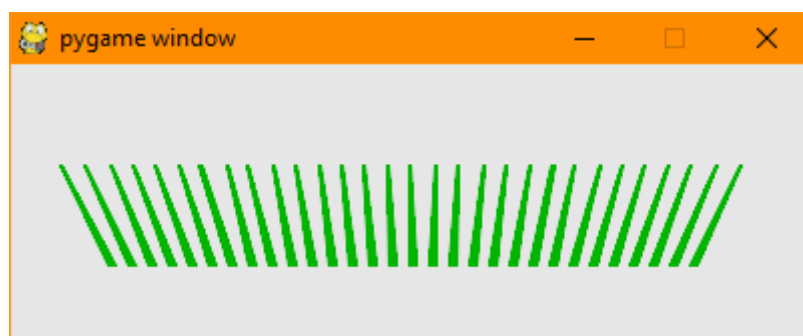
draw_repeatedly( draw_grass )
Ln: 202 Col: 0

```

11. Relate the description below to the changes in the code.

- dx moves the tips of the grass left or right.
- randint(-10, 10) is a function from the random library that generates a random integer between -10 and +10. This is then divided by 10 to get a floating-point number between -1 and +1. So, the randint() is moving the tips of the grass one pixel randomly left or right.
- The 2*t*(i-15) isn't random. It makes the tips of the grass spread out. If this part of the expression is left out from the code the tips will just move randomly, but they won't spread out.

12. Run the program to get the following picture:



The printed picture doesn't show that the grass is now moving, but does show how it has spread out.

13.7 REVIEW EXERCISE

1. An event is:
 - a. When a program is running and moves from code written by the programmer into library code.
 - b. Something that happens that a program should react to.
 - c. When a program has too much data and needs to get rid of some of the data.
 - d. A special Python data type used to track important dates, for example in calendar apps.

2. The function 'randint' is found in which library?
 - a. The math library.
 - b. The randint library.
 - c. The monster library.
 - d. The random library.

3. The string named greeting has the value 'Good'. Which command sets it to have the value 'Good Morning'?
 - a. `greeting = 'Good' + greeting`
 - b. `greeting = greeting + " Morning"`
 - c. `greeting = 'Good' + 'Morning'`
 - d. `greeting = ['Good', 'Morning']`

4. The tuple named colour has the value (0,0,255). Which command sets it to have the value (255,0,255)?
 - a. `colour[0] = 255`
 - b. `colour[1] = 255`
 - c. `colour = (255 , 0+0 , 255)`
 - d. `colour = 255,0,255`

5. Which of the following does not generate events in Python with Pygame?
 - a. Clicking on the mouse.
 - b. A timer.
 - c. Input from keyboard.
 - d. Battery low.

LESSON 14 – RECURSION

After completing this lesson, you should be able to:

- Understand the term recursion

This lesson also reinforces concepts from earlier lessons such as:

- Loop
- Procedure
- Function
- Variable
- Parameter

14.1 RECURSION



Concepts

The waving grass example from lesson 13 and the arrow example from lesson 11 involved subroutines calling other subroutines. This is one of the ways we decompose a problem in computing. Remember decomposition is about breaking a problem down into smaller problems. Subroutines are written to solve these smaller problems. Subroutines are then called to execute until the whole problem is solved.

It is also possible for a subroutine to call itself one or many times. Provided there is some safeguard against a subroutine calling itself forever, subroutines that call themselves can be a powerful technique for decomposing certain problems.

A subroutine that calls itself is known as Recursion.

Recursion is the process of a subroutine dividing a problem into simpler parts and calling itself to solve those simpler parts. A recursive function has a call to itself within the function.

When a recursive function invokes itself it is said to **recurse**.

The technique of recursion is most useful when a problem can be decomposed into two smaller problems that are still in many ways like the original problem.

Example: Searching

One example of solving a problem using recursion is when searching. A subroutine for searching could split the possibilities being searched into two areas. The subroutine could invoke itself to search each of the areas, and split those, and so on.

Imagine searching a house. The algorithm would split the problem into searching the rooms and then split searching the rooms into searching different areas in the room. Eventually the area being searched becomes small enough that it can be checked in one step. There is then no need to subdivide any further.

Example: Drawing a Curved Line

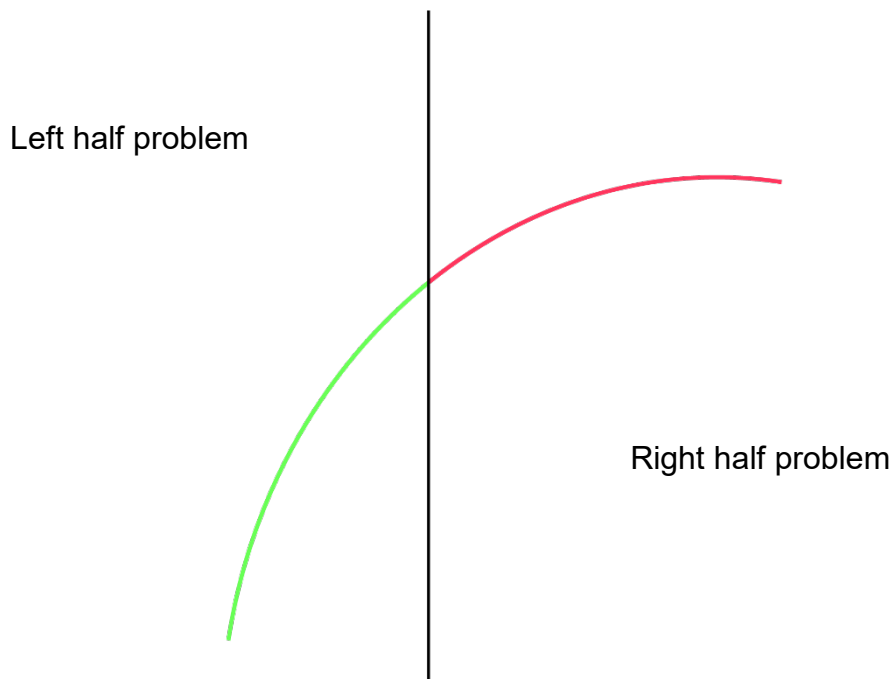
Another example of a problem that can be solved with recursion is that of drawing a curved line:

An algorithm chooses a point half way along the line. That splits the curved line into a left part and a right part.

Splitting the curve has decomposed the original problem into two smaller problems. One problem is drawing the left half of the curved line, the other problem is drawing the right half.

That might not seem like much of an improvement in efficiency, but it can be, if we further split each half of the curved line into two smaller problems, and so on.

By repeatedly subdividing the problem, the part of the curved line to draw eventually becomes small enough that it fits in just one dot, one pixel on the screen.



How the program works:

- A recursive subroutine to draw a curved line can call itself to draw the left and the right halves.
- It can have a logic test that detects when the problem is a pixel or smaller in size.
- When the problem has been reduced that far, the subroutine can draw that dot rather than doing more subdividing.
- The combined result of all the dots form the curved line.

14.2 RECURSIVE DRAWING

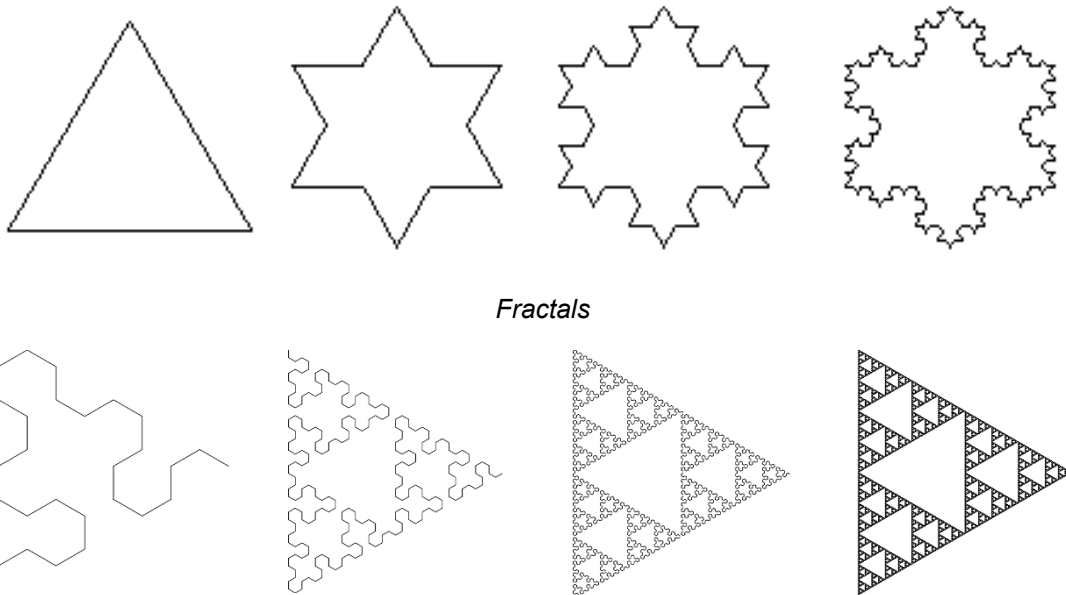
Concepts

The curved line drawing algorithm recurses once to draw the left part, once to draw the right part. Recursion does not always have to split a problem into two, it can split a problem into many parts. The upcoming example to draw a fern leaf splits the problem into four at each stage.

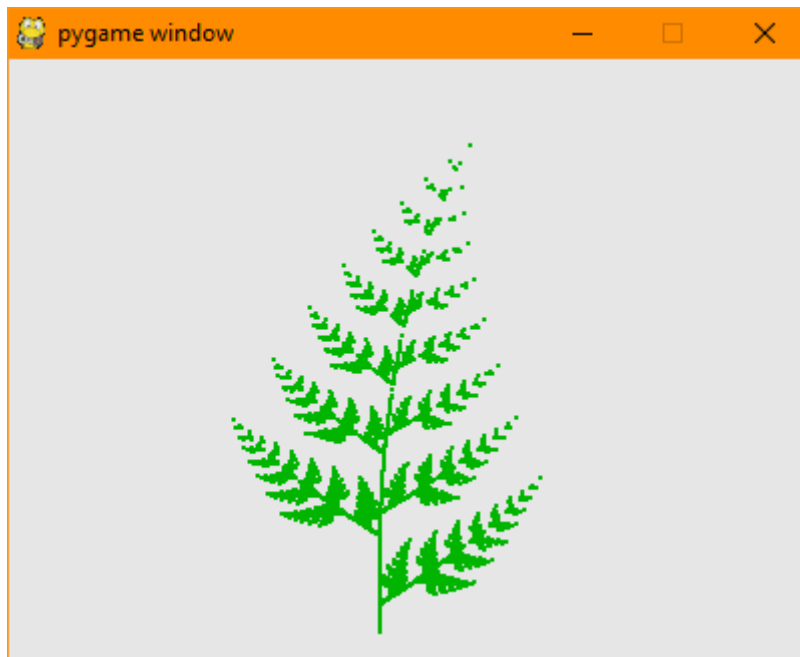
Let's first take a quick look at a fractal before we continue.

A **fractal** is a shape that is similar to itself at different scales. If you zoom in to a fractal, you can find patterns that are similar to the patterns of the whole shape.

Some examples of how a fractal develops are shown below:



The next example uses recursion to draw a **fractal** that is a fern leaf.



How the program works:

- The fractal drawing algorithm in the example below recurses to draw four different parts.
- The four different choices make smaller distorted versions of the whole pattern in four different places. In turn four smaller distorted versions are made in each of those places.
- The details of the four distortions are held as numbers in list variables called f1, f2, f3, and f4.
- Often recursive code has a parameter that sets how far to recurse. In this fractal example this parameter is called 'level'. Each time the function recurses it decreases the value of level. When level reaches zero the algorithm plots a dot, and stops subdividing the problem further.

Example: Recursion for Fractal Fern

1. Make a copy of the boilerplate code
2. Call it Fern.py
3. Add the following code:

```

76 Fern.py - C:/Python32/Fern.py
File Edit Format Run Options Windows Help

# Define four 'affine transformations'
f1 = [0,0,0,0.22,0,0]
f2 = [0.85,0.04,-0.04,0.85,0,1.6]
f3 = [0.2,-0.26,0.23,0.22,0,1.6]
f4 = [-0.15,0.28,0.26,0.24,0,0.44]

# The recursive function which calls itself.
# It calls itself four times
# Each of those in turn calls itself four times,
def draw_fern( level, p ):
    if level < 1 :
        plot_point( p )
        return
    draw_fern( level-1, apply_transform( f1, p ) )
    draw_fern( level-1, apply_transform( f2, p ) )
    draw_fern( level-1, apply_transform( f3, p ) )
    draw_fern( level-1, apply_transform( f4, p ) )

def draw_the_fern():
    draw_fern( 9, (0,0) )

draw_once( draw_the_fern )

Ln: 205 Col: 0

```

4. Run the program to get the result.



5. Check that the run has finished.

```

>>> ===== RESTART =====
>>>
About to draw
Finished

```

Example: Modifying Recursion Code for Fractal Fern

We can modify the shape of the fern by changing some parameters in the recursion code.

```
def draw_the_fern():  
    draw_fern( 9, (0,0) )
```

If you reduce the 9 in `draw_fern(9, (0,0))` to 3 the program will plot far fewer points.

Increasing the '9' even a little bit, will make the program take a lot longer to run.

If you change the 9 to 30 the program will take over a million years to complete running.

Tip: To stop the program before it finishes, use the same technique as for the infinite loop. Go to the shell window and on the menu click 'Shell', and then 'Restart Shell'.

14.3 REVIEW EXERCISE

1. A recursive algorithm is one that:
 - a. Draws a picture of a fern.
 - b. Draws a curved line.
 - c. Uses random numbers to solve a problem.
 - d. Breaks a problem into smaller pieces, and then applies the same approach to those smaller pieces.

2. A function that calls itself:
 - a. Is an incorrect sequence error in Python.
 - b. Could be an example of recursion.
 - c. Will always cause an infinite loop.
 - d. Is called a distortion.

3. An infinite loop is:
 - a. An example of recursion.
 - b. An example of distortion.
 - c. A loop that runs forever.
 - d. A recursive function that draws circles.

LESSON 15 – TESTING AND MODIFICATION

After completing this lesson, you should be able to:

- Understand types of errors in a program like: syntax, logic
- Understand the benefits of testing and debugging a program to resolve errors
- Identify and fix a syntax error in a program like: incorrect spelling, missing punctuation
- Identify and fix a logic error in a program like: incorrect Boolean expression, incorrect data type
- Check your program against the requirements of the initial description
- Identify enhancements, improvements to the program that may meet additional, related needs
- Describe the completed program, communicating purpose and value

15.1 TYPES OF ERRORS

Concepts

A flowchart or a program may have errors. Errors which make the algorithm or program work incorrectly are called **bugs**. For example:

- If one of the decision boxes in a flowchart was labelled incorrectly, with the 'Yes' and 'No' in the wrong places, the algorithm would have an incorrect output. We call this type of error a **bug**.
- If in the magic trick program we divided by 15 rather than 13, we wouldn't get the intended answer at the output. The step dividing by 15 would be called a **bug** in the program.

Serious bugs cause programs to stop working, or **crash**.

A **crash** is when the program stops working completely. Users can lose their data as the result of a crash.

Three types of error commonly occur when programming:

A **syntax error** occurs when a construct in the programming language is incorrectly written. It is an error in the code due to incorrect use of the language such as spelling errors, incorrect punctuation, incorrect naming and referencing. For example, (A+3 is a syntax error because it is missing a closing bracket.

A **logic error** occurs when the program instruction is for an incorrect action. The program appears to be syntactically correct, (i.e. written correctly) but the logic is flawed so when it runs it doesn't do what was expected. In other words, a program may operate correctly but may not do what is required.

For example:

```
if test_tube_is_full :  
    pour_more_acid_into_it().
```

A **datatype** error or **typeerror** occurs when you attempt to use an operator or a function on something of the wrong type. For example, if we try to multiply a string and an interger **3*apples** together.

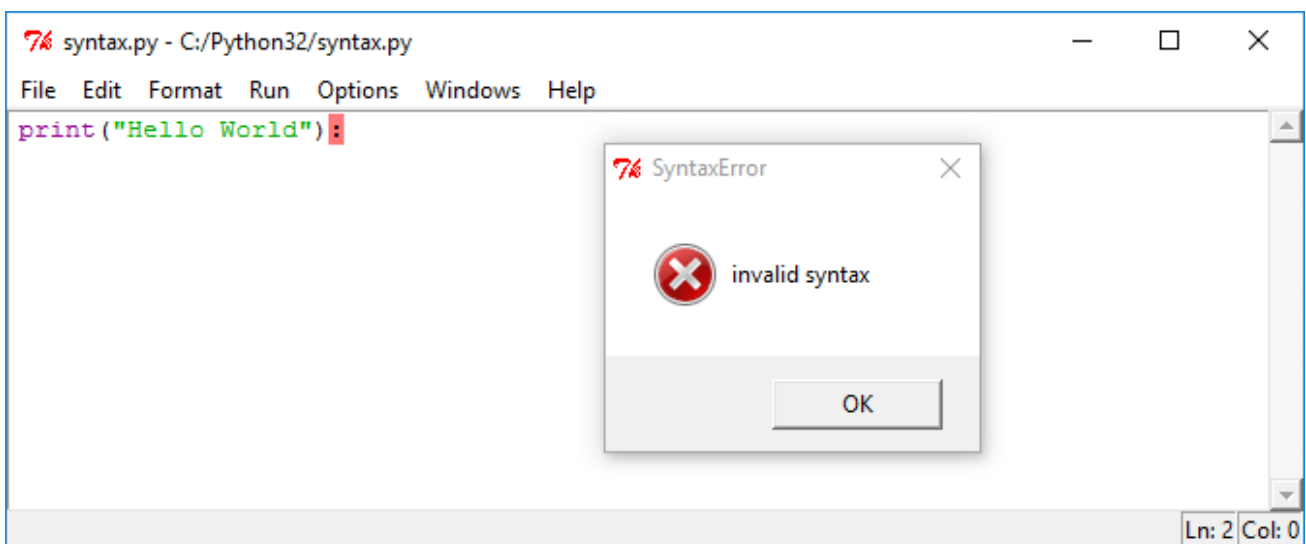
15.2 FINDING ERRORS

Concepts

Syntax Errors

Syntax errors also known as parsing errors are detected immediately by Python when you attempt to run a program. Syntax errors aren't thought of as bugs, since Python will prevent the program from running when it finds a syntax error.

Examples of syntax errors include incorrect syntax, omission of a required colon or bracket, misspelling a keyword, incorrect format for numbers etc. The parser highlights the earliest point in the line where an error is detected



There is a colon at the end of the line which is incorrect. Simply correct this error by removing the colon and re-run the program.

```

syntax.py - C:\Python32\syntax.py
File Edit Format Run Options Windows Help
print("Hello World")
Ln: 1 Col: 20

```

```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Hello World
>>>
Ln: 6 Col: 4

```

Logic Errors

Logic errors are not always easy to find. Consider the logic in this program to display an instruction to put a cake in an oven when the oven reaches the correct temperature.

```

Oven_Temp.py - C:/Python32/Oven_Temp.py
File Edit Format Run Options Windows Help
Correct_Temp="180C"
Current_Temp= "75C"
if Current_Temp!=(Correct_Temp):
    print("Put Cake in Oven")
else:
    print("Wait until the oven is at the correct temperature")
Ln: 3 Col: 32

```

```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Put Cake in Oven
>>>
Ln: 6 Col: 4

```

The logic is wrong . To correct we need to change the != (Not equal) symbol to the == (Equals) symbol.

```
Oven_Temp.py - C:/Python32/Oven_Temp.py
File Edit Format Run Options Windows Help
Correct_Temp="180C"
Current_Temp= "75C"
if Current_Temp==(Correct_Temp):
    print("Put Cake in Oven")
else:
    print("Wait until the oven is at the correct temperature")
Ln: 3 Col: 25
```

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Wait until the oven is at the correct temperature
>>>
Ln: 6 Col: 4
```

Consider the example below and see if you can correct the logic:

```
Age.py - C:/Python32/Age.py
File Edit Format Run Options Windows Help
Age = input("How old is he?")
Age = int(Age)
if Age > 13 and Age < 19:
    print("He's a teenager")
else:
    print("He's not a teenager")
Ln: 1 Col: 18
```

There is a problem with the logic test in this program.

The logic test will work correctly for age inputs 14, 15, 16, 17 and 18 as the program will correctly print “He’s a teenager” for each input.

But for Age inputs 13 and 19 it will print “He’s not a teenager”

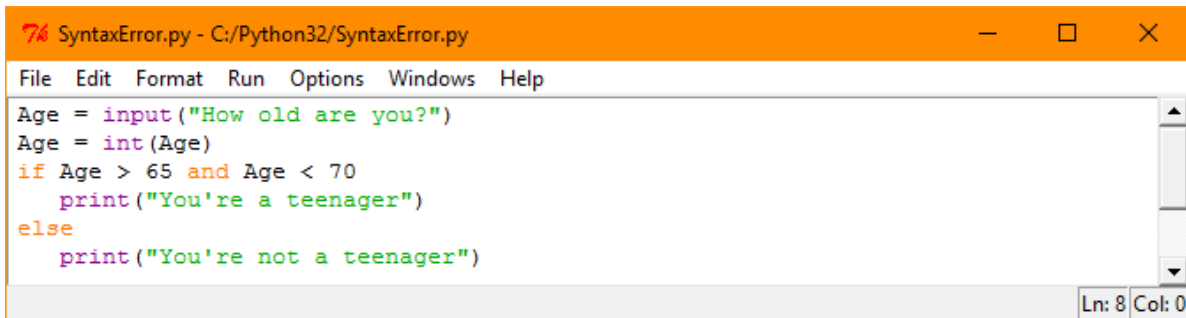
To fix the program:

1. Open Python.
2. Type the code above correctly to fix the bug, by changing the line of code beginning with ‘if’.
3. Check your solution by running your edited program.
4. Test values for age such as 12, 13, 14, 18, 19 and 20 to see if you get the expected output.

Example: Find and Fix logic and syntax errors

The following program has two syntax errors and one logic error.

1. Type in the program as shown below.

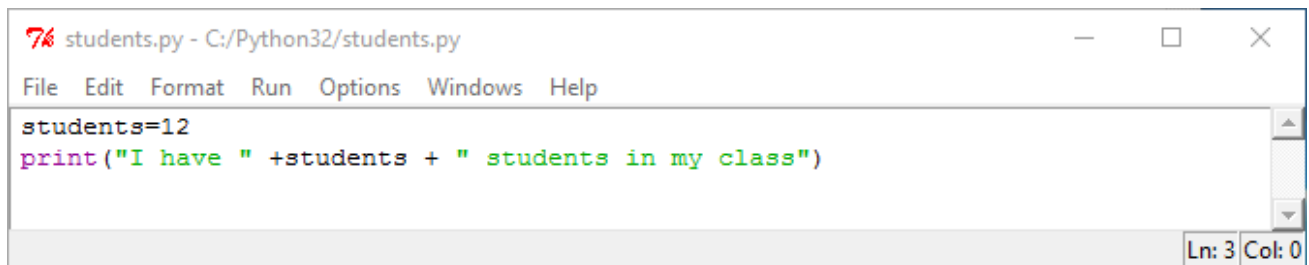


```

SyntaxError.py - C:/Python32/SyntaxError.py
File Edit Format Run Options Windows Help
Age = input("How old are you?")
Age = int(Age)
if Age > 65 and Age < 70
    print("You're a teenager")
else
    print("You're not a teenager")
Ln: 8 Col: 0
  
```

2. Fix the syntax errors.
3. Fix the logic error too.
4. Run the program to check your work.

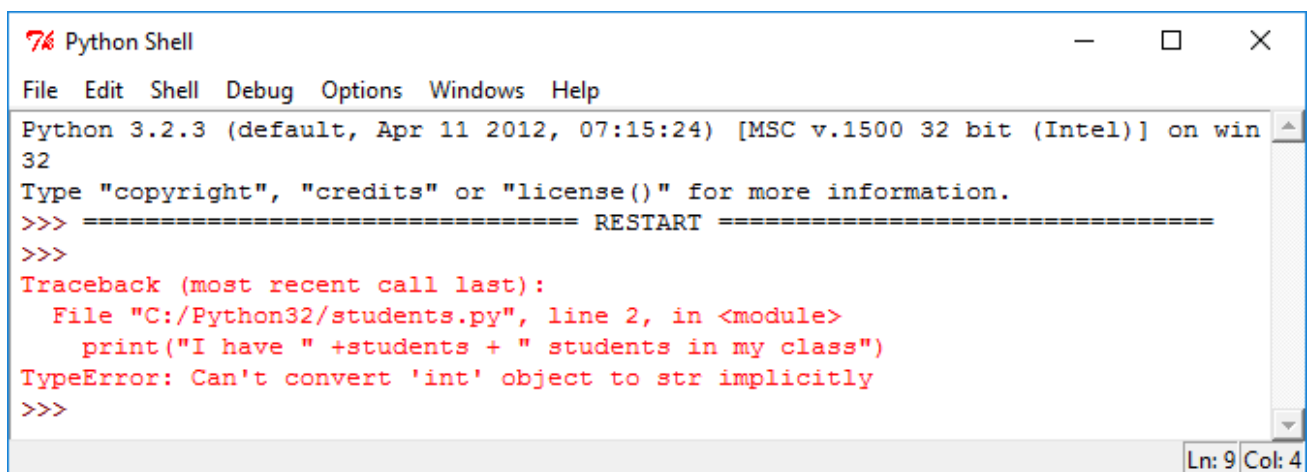
Example: Find and Fix datatype error



```

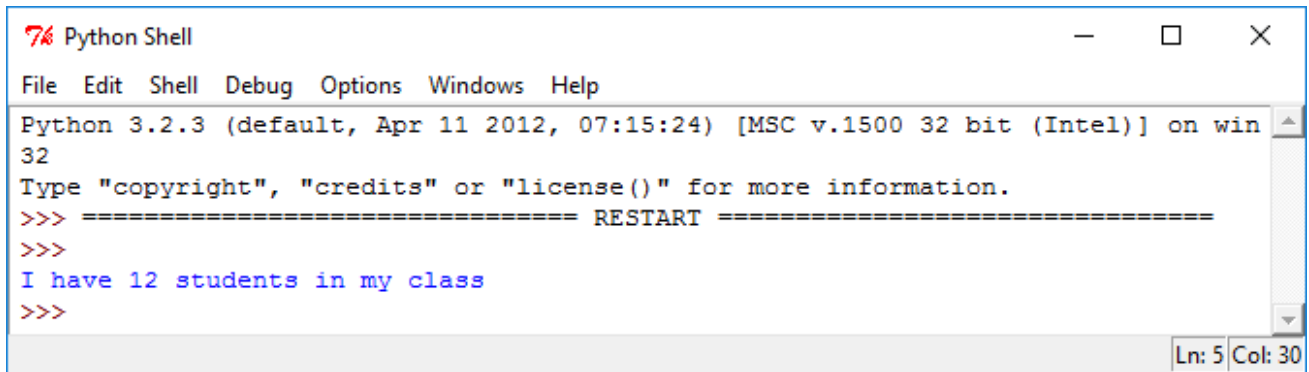
students.py - C:/Python32/students.py
File Edit Format Run Options Windows Help
students=12
print("I have " +students + " students in my class")
Ln: 3 Col: 0
  
```

Here we cannot join different datatypes together. We cannot join strings to number variables. However, we can convert a number variable to a string. Below, by using the `str()` function we convert the number variable `students` into a string.



```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Traceback (most recent call last):
  File "C:/Python32/students.py", line 2, in <module>
    print("I have " +students + " students in my class")
TypeError: Can't convert 'int' object to str implicitly
>>>
Ln: 9 Col: 4
  
```



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.2.3 (default, Apr 11 2012, 07:15:24) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
I have 12 students in my class
>>>
Ln: 5 Col: 30
```

15.3 TESTING AND DEBUGGING A PROGRAM



Concepts

Checking for Errors

A programmer might notice that their program has a logic error, or they might not. It is important to detect logic errors. One way to do this systematically is by **testing**.

Testing

Running a computer program using different inputs to find if there are logic errors.

In testing, the tester needs to know what the results should be. They may refer to the requirements in a specification to check that the program is doing what it is supposed to.

After fixing a bug the program will be tested again to make sure the bug really has been fixed.

Testing Against Requirements

Best practice is to test a program against a specification document written at the design stage. The specification gives clarity about what the program should do. It lists requirements for the program.

A specification for a program will usually number the requirements. This is a possible example:

1.3.1: The program SHALL dim the screen if the computer has not been used for one minute.

A corresponding plan for testing would include a step for testing that requirement. The plan would stipulate leaving the program running for one minute and confirming that the screen does go dim after that time. Each requirement should have some corresponding tests.

In this example, if after one minute the program crashes, there is a bug in the program. However, if nothing special happens after a minute, one could argue that there is no bug.

This shows how systematic testing against requirements is valuable. Testing would have identified a requirement that had not been implemented. Testing against requirements also helps in finding problems that only show when certain parts of the program are used. Testing in this way may find problems that would not otherwise have been found.

Finding Bugs

When a program gives the wrong or unexpected results, or if the program crashes unexpectedly with an error, it is not always obvious why that has happened.

When a program gives the wrong answer, it may not always be enough to just look at the code to find the bug.

One way to find out more about the bug is to include some extra diagnostics in the code. These could be additional print statements in the program to display the values of variables at various points in the program. These extra print statements show how the values were changing.

The work of finding out the cause of a bug is called **debugging**.

Debugging

Running a faulty program with extra diagnostic information to work out where the mistakes in the code are, and then fix those mistakes.

One other way to diagnose a program is to run it 'under a debugger'. This is a tool that can be used to run the program one line at a time. Python has a debugger that can be enabled from the Python shell window.

Debugging may uncover the bug and also show when the bug first occurs. Without debugging, it may not be clear where the problem code is.

15.4 IMPROVING A PROGRAM



Concepts

Once a program is written and is successful, there is often pressure to create a new and improved version of it.

Commercial considerations usually mean that a complete rewrite of the program is not a good idea. A better plan is usually to make small incremental improvements that make the program more useful to more people. Such changes help the code to solve a more general problem than the initial version.

Suitability for Different Countries

Recall that in the lesson on computational thinking we looked at generalisation. We saw that a washing machine design might be generalised by making it suitable for more countries. This is very common in software too.

Software may benefit from translation of the strings into other languages. Usually programmers attempt to use the same source code, but make the code more flexible so that it can use different strings for different languages as indicated by a parameter.

Generalising code to make it suitable for different countries is sometimes called **localisation**. Localisation can go deeper than simply translation. A recipe program might support different kinds of measurement. For example, a recipe program might support temperature measured in degrees Fahrenheit or in degrees Celsius. This would help it to be useful in more countries.

Work on localisation may also draw attention to other related opportunities for generalisation.

Localisation work on providing quantities of ingredients in different measurement units, as needed in different countries, could lead on to other useful generalisations. An improved program could calculate quantities in different units. It could allow the user of the program to request adjustments to portion size and number of portions, with the program recalculating the quantities.

Solving a Larger Problem

In the lesson on computational thinking recall that we saw that a solution to the problem of running a musical festival could be generalised. Experience in running a music festival could contribute to solving the more ambitious problem of running a musical tour.

In looking for opportunities to improve a recipe program, small additions that, for example, count calories, or add up nutritional information, could generalise the recipe program and make it part of a larger computer program for health and nutrition.

Successful improvement to a program needs to balance ambition with practicality. A large change in functionality of a program may require a rewrite, or may require a new library to bring in new functionality. Large changes, such as adding voice recognition into a program go beyond what can be achieved with small incremental improvements.

Presenting a Program

A programmer's view of a program is very different to the view of the program a user or a manager needs to have. When presenting a program, be aware of the audience and what aspects of the program are relevant to them. This is particularly important when communicating proposed improvements. Pseudocode and flowcharts may be relevant in showing how an algorithm works. They do little to communicate purpose and value.

Communicating Purpose: When communicating the purpose of a program, explain what problem the program solves. Generally programs are useful in parts of a problem that are repeated again and again. Computers are good at automation.

In communicating purpose, it may be useful to also communicate what parts of a problem a program does *not* solve.

A word processing program saves an author from typing the same text out many times as they work through different versions of a document. A word processing program does little to help an author find the right words and find ideas for characters and plot.

Communicating Value: The value of a program or improvements to a program depend on who will use it in the market and on competition. Managers may want to know if a similar program exists already, or if the new program can easily be copied by others.

If the algorithms are new, creative, innovative, it may be possible to patent them and so protect them from being copied.

If the program uses a proprietary file format, i.e. owned by the programmer or his/her company, that stores data in a way that other programs can't easily read, that may inhibit other companies from making their own version.

In communicating the value of a program, highlight what is unique about it. Show that the unique aspects are of value to end users. A valuable program is one that saves the end user time and money. Where possible support your claims about the value of a program with numbers.

15.5 REVIEW EXERCISE

1. Which expression has a syntax error?
 - a. $2+4+8$
 - b. $(2+4)*8$
 - c. $2/4/8$
 - d. $2+(4*8))$

2. A logic test was intended to test if variable c, d and e are in order with c smallest and e largest. Which of the following has a logic error:
 - a. if $(c<d)$ and $(d<e)$:
 - b. if $(e>d)$ and $(d>c)$:
 - c. if $(c\#d)$ and $(d\#e)$:
 - d. if $(e<d)$ and $(d<c)$:

3. Program testing is:
 - a. Checking with the customer that the specification is what they want.
 - b. Fixing a bug when a user reports an error.
 - c. When someone who has more experience has a look at the source code for a program and helps improve it.
 - d. A formal approach to finding bugs in a program, by running the program with different inputs.

4. Debugging is:
 - a. Checking a program against its requirements to see if it does what it is supposed to do.
 - b. Enhancing a program by adding new features.
 - c. Work done to track down the cause of a problem and fix it.
 - d. Enhancing a program by taking out features which aren't needed anymore.

5. One of the initial requirement for a particular program is that it ask for and remember your name. Which of the following would you expect to see in the code to help meet that requirement?
 - a. A recursive function
 - b. An `input()` statement
 - c. A bug
 - d. Code to convert a string to an integer.

6. A software company wants to make incremental improvements to version 1 of its program for managing recipes. Which of the following is a good way to identify possible enhancements?
 - a. Add code for drawing a fern recursively.
 - b. Use Google to search for "robots AND cooking"
 - c. Look for small changes that would make the program useful to more people.

7. When describing the commercial value of a new program, the best way is:
- a. Show as detailed a flowchart as possible to show how complex the program is.
 - b. Describe the benefit to users and back it up with numerical data if you can.
 - c. Explain that the program is new, and therefore it must be of value.
 - d. Explain that it uses only well known, widely used, reliable algorithms.

ECDL Syllabus

Ref.	Task Item	Location	Ref.	Task Item	Location
1.1.1	Define the term computing.	1.1 <i>Computational Thinking</i>	2.1.4	Use abstraction to filter out unnecessary details when analysing a problem.	1.1 <i>Computational Thinking</i>
1.1.2	Define the term computational thinking.	1.1 <i>Computational Thinking</i>	2.1.5	Understand how algorithms are used in computational thinking.	1.1 <i>Computational Thinking</i> 1.2 <i>Instructing a Computer</i>
1.1.3	Define the term program.	1.2 <i>Instructing a Computer</i>	2.2.1	Define the programming construct term sequence. Outline the purpose of sequencing when designing algorithms.	3.1 <i>Steps in an Algorithm</i>
1.1.4	Define the term code. Distinguish between source code, machine code.	2.2 <i>Computer Languages</i>	2.2.2	Recognise possible methods for problem representation like: flowcharts, pseudocode.	3.1 <i>Steps in an Algorithm</i> 3.2 <i>Flowcharts</i>
1.1.5	Understand the terms program description and specification.	2.3 <i>Text About Code</i>	2.2.3	Recognise flowchart symbols like: start/stop, process, decision, input/output, connector, arrow.	3.2 <i>Flowcharts</i>
1.1.6	Recognise typical activities in the creation of a program: analysis, design, programming, testing, enhancement.	2.4 <i>Stages in Developing a Program</i>	2.2.4	Outline the sequence of operations represented by a flowchart, pseudocode.	3.2 <i>Flowcharts</i> 3.3 <i>Pseudocode</i>
1.1.7	Understand the difference between a formal language and a natural language.	2.1 <i>Precision of Language</i>	2.2.5	Write an accurate algorithm based on a description using a technique like: flowchart, pseudocode.	3.2 <i>Flowcharts</i> 3.3 <i>Pseudocode</i>
2.1.1	Outline the typical methods used in computational thinking: decomposition, pattern recognition, abstraction, algorithms.	1.1 <i>Computational Thinking</i>	2.2.6	Fix errors in an algorithm like: missing program element, incorrect sequence, incorrect decision outcome.	3.4 <i>Fixing Algorithms</i>
2.1.2	Use problem decomposition to break down data, processes, or a complex problem into smaller parts.	1.1 <i>Computational Thinking</i>			
2.1.3	Identify patterns among small, decomposed problems.	1.1 <i>Computational Thinking</i>			

Ref.	Task Item	Location	Ref.	Task Item	Location
3.1.1	Describe the characteristics of well-structured and documented code like: indentation, appropriate comments, descriptive naming.	9.1 <i>Readable Code</i> 9.2 <i>Comments</i> 9.3 <i>Organisation of Code</i> 9.4 <i>Descriptive Names</i>	3.2.4	Use appropriately named variables in a program for calculations, storing values.	6.2 <i>Variables</i> 9.4 <i>Descriptive Names</i>
3.1.2	Use simple arithmetic operators to perform calculations in a program: +, -, /, *.	5.1 <i>Performing Calculations with Python</i>	3.2.5	Use data types in a program: string, character, integer, float, Boolean.	6.1 <i>Data Types</i> 6.2 <i>Variables</i> 6.3 <i>Beyond Numbers</i> 7.4 <i>Booleans and Variables</i>
3.1.3	Understand the precedence of operators and the order of evaluation in complex expressions. Understand how to use parenthesis to structure complex expressions.	5.2 <i>Precedence of Operators</i> 7.5 <i>Putting it all together</i>	3.2.6	Use an aggregate data type in a program like: array, list, tuple.	8.1 <i>Aggregate Data Types</i> 8.2 <i>Lists</i> 8.3 <i>Tuples</i>
3.1.4	Understand the term parameter. Outline the purpose of parameters in a program.	11.3 <i>Procedures and Functions</i>	3.2.7	Use data input from a user in a program.	6.3 <i>Beyond Numbers</i> 10.3 <i>IF ELSE Statement</i>
3.1.5	Define the programming construct term comment. Outline the purpose of a comment in a program.	9.2 <i>Comments</i>	3.2.8	Use data output to a screen in a program.	4.2 <i>Exploring Python</i> 6.2 <i>Variables</i>
3.1.6	Use comments in a program.	9.2 <i>Comments</i>	4.1.1	Define the programming construct term logic test. Outline the purpose of a logic test in a program.	10.2 <i>IF Statement</i>
3.2.1	Define the programming construct term variable. Outline the purpose of a variable in a program.	6.2 <i>Variables</i>	4.1.2	Recognise types of Boolean logic expressions to generate a true or false value like: =, >, <, >=, <=, <>, !=, ==, AND, OR, NOT.	7.2 <i>Comparison Operators</i> 7.3 <i>Boolean Operators</i> 7.5 <i>Putting it all together</i>
3.2.2	Define and initialise a variable.	6.2 <i>Variables</i>	4.1.3	Use Boolean logic expressions in a program.	10.2 <i>IF Statement</i> 10.3 <i>IF ELSE Statement</i>
3.2.3	Assign a value to a variable.	6.2 <i>Variables</i>	4.2.1	Define the programming construct term loop. Outline the purpose and benefit of looping in a program.	12.1 <i>Looping</i>

Ref.	Task Item	Location	Ref.	Task Item	Location
4.2.2	Recognise types of loops used for iteration: for, while, repeat.	12.1 <i>Looping</i> 12.2 <i>Looping with Variables</i> 12.3 <i>Variations on Loops</i>	4.5.1	Understand the term event. Outline the purpose of an event in a program.	13.2 <i>Events</i>
4.2.3	Use iteration (looping) in a program like: for, while, repeat.	12.1 <i>Looping</i> 12.2 <i>Looping with Variables</i> 12.3 <i>Variations on Loops</i> 12.4 <i>Putting it all together</i>	4.5.2	Use event handlers like: mouse click, keyboard input, button click, timer.	13.4 <i>Pygame Library</i> 13.6 <i>Drawing Using the Libraries</i>
4.2.4	Understand the term infinite loop.	12.3 <i>Variations on Loops</i>	4.5.3	Use available generic libraries like: math, random, time.	13.2 <i>Standard Libraries</i>
4.2.5	Understand the term recursion.	14.1 <i>Recursion</i>	5.1.1	Understand the benefits of testing and debugging a program to resolve errors.	15.3 <i>Testing and Debugging a Program</i>
4.3.1	Define the programming construct term conditional statement. Outline the purpose of conditional statements in a program.	10.2 <i>IF Statement</i>	5.1.2	Understand types of errors in a program like: syntax, logic.	15.1 <i>Types of Errors</i>
4.3.2	Use IF...THEN...ELSE conditional statements in a program.	10.2 <i>IF ELSE Statement</i>	5.1.3	Run a program.	4.3 <i>Save a Program</i>
4.4.1	Understand the term procedure. Outline the purpose of a procedure in a program.	11.3 <i>Functions and Procedures</i>	5.1.4	Identify and fix a syntax error in a program like: incorrect spelling, missing punctuation.	15.2 <i>Finding Errors</i> 15.3 <i>Testing and Debugging a Program</i>
4.4.2	Write and name a procedure in a program.	11.3 <i>Functions and Procedures</i> 12.4 <i>Putting it all together</i>	5.1.5	Identify and fix a logic error in a program like: incorrect Boolean expression, incorrect data type.	15.2 <i>Finding Errors</i> 15.3 <i>Testing and Debugging a Program</i>
4.4.3	Understand the term function. Outline the purpose of a function in a program.	11.3 <i>Functions and Procedures</i>	5.2.1	Check your program against the requirements of the initial description.	2.4 <i>Stages in Developing a Program</i> 15.3 <i>Testing and Debugging a Program</i>
4.4.4	Write and name a function in a program.	11.3 <i>Functions and Procedures</i> 12.4 <i>Putting it all together</i>	5.2.2	Describe the completed program, communicating purpose and value.	15.4 <i>Improving a Program</i>
			5.2.3	Identify enhancements, improvements to the program that may meet	15.4 <i>Improving a Program</i>

Ref.	Task Item	Location
	additional, related needs.	

Congratulations! You have reached the end of the ECDL Computing book. You have learned about the skills and competences relating to computational thinking and coding that will guide you through the process of problem solving and creating simple computer programs:

- Understand key concepts relating to computing and the typical activities involved in creating a program.
- Understand and use computational thinking techniques like problem decomposition, pattern recognition, abstraction and algorithms to analyse a problem and develop solutions.
- Write, test and modify algorithms for a program using flowcharts and pseudocode.
- Understand key principles and terms associated with coding and the importance of well-structured and documented code.
- Understand and use programming constructs like variables, data types, and logic in a program.
- Improve efficiency and functionality by using iteration, conditional statements, procedures and functions, as well as events and commands in a program.
- Test and debug a program and ensure it meets requirements before release.

Having reached this stage of your learning, you should now be ready to undertake an ECDL certification test. For further information on taking this test, please contact your ECDL test centre.

ECDL Malta

Flat 2, St John's Flats,
Naxxar Road,
Birkirkara BKR9040
Email: info@ecd.com.mt

Phone: +356 2166 7706

Web: www.ecdl.com.mt