YEAR 12 PROGRAMMING & PROBLEM SOLVING

Student Workbook in Python

WRITTEN BY

Sandy Garner and Anthony Robins

Department of Computer Science

University of Otago

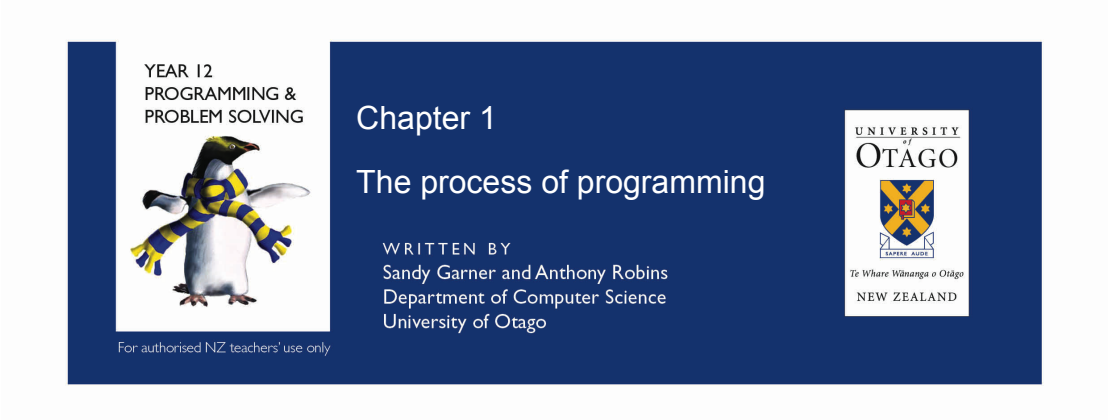
Contents:

Chapter 5 Boolean and Selection

Chapter 6 Repetition

Chapter 7 Lists

Chapter 8 Strings



1.0 Introduction

The essence of programming is **problem solving**. Before you can write a program you need to understand what the problem is. Then you need to be able to describe a solution to the problem very precisely. A sufficiently precise description of a process is called an **algorithm**. So you need to work out an algorithm that solves the problem. That's the hard work done!

It is always helpful to design the solution to a problem using pencil and paper. A precise solution can be described using a diagram called a **flowchart**. Sometimes it helps the processes of both problem solving and programming to develop algorithms in some informal language that follows the structural conventions of a programming language – such a language is called **pseudocode**.

Whether you used a flowchart, pseudocode or some other description of your algorithm, the next step is to write the algorithm down in a language that a computer can understand – a **programming language**. In other words, you need to turn the algorithm into a **program** (sometimes called computer "code"). This process is sometimes called **coding**.

There are many **programming languages**, and the particular one that you choose is usually not critical. It is the algorithm which is the solution to the problem, the program is just a way of writing it down for a computer. Having said that, different languages have different strengths and weaknesses, and make different things easy or hard to write.

1.1 The process

There are many descriptions of the correct process to follow when writing a program. This is a typical example.

You are presented with a problem, and need to devise a solution:

- Clarify the problem:
  - identify the input – the data (information) that is required by the problem, and where it comes from
  - identify the output – the data or actions that are required and how they should be presented
- Design a solution:
  - plan a series of steps which will take you from input to output
  - refine this plan until it is completely precise, an algorithm.
- Code the algorithm as a program
- Test the program
- If necessary, improve the program design, code and test again
- Document the program

The first two steps do not require a computer. This is the program design stage.

Documentation is listed last in our sequence but you should document as you code and test.

1.2 Introducing the tools

Tools

As described above, a **programming language** is a way of writing down an **algorithm** in a form that can be processed by a computer. Modern programming languages are *high level* languages which are supposed to be easy for humans to understand. The basic elements of a program are the individual **statements**. Each statement is an instruction which might be describing data or an operation to be performed.In Python there is usually one statement per line of the program. If there are multiple statements on a line then they must be separated by a semi-colon `";"`.

As a word processor is a tool for creating documents, so an **IDE** (integrated development environment) is a tool for creating programs. An IDE helps with writing program code by such things as colouring the key words of the language, formatting the code, numbering the lines, highlighting blocks of code, and so on.

An IDE uses other programs to process the high level program code that we write.Python code is processed by an **interpreter** which creates machine code which runs / executes on the particular computer or device. The interpreter will give useful error messages if the program code does not follow the rules of the language (a **syntax error**) or cannot run (a **run-time error**).

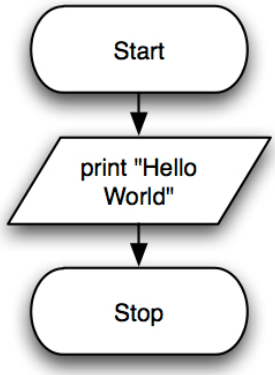
Hello World

In keeping with time honoured tradition, our first task is to write "Hello World" on the display screen.

A flowchart describing the algorithm is shown below. A flowchart has a start point and a stop point. Each process of the program solution is described in a box and the design uses arrows to flow from start to stop.

The pseudocode for this task is simply:

```
print "Hello World"
```



Hello World Flowchart

Coding the solution:

In your IDE make a new file (in Idle "New window") and in it type:

```
print("Hello World")
```

Save the file and give it a name (such as Hello.py). The file name should end in ".py" to indicate that it is a Python program. File names should be meaningful and describe what the program is about where possible.

That's it! Your program is written. You can run your program (in Idle, use the F5 key). One of the nice things about Python is that programs do not have any required "setting up" code the way most languages do. Consequently simple programs are simple to write.

> ***Note:*** *You can type a Python program directly in to the Python shell, but in this Workbook we will type them into separate named files so that they can be saved.*

In this example, we don't need to know about the inner workings of the computer to send data to and place it in order on the display. This is such a common thing to want to do that Python comes with a built in function which will print a line of text to the display. That function is used in the program above.

If you made any errors in typing this program you would have got some kind of error message when you tried to run it. Try this by removing one of the quote marks (you get a syntax error message) or by misspelling print (you get a run-time error message). Python is case sensitive, so Print is not the same as print.

Names

Many parts of a program, including variables (see 1.3 below), need to be named. In programming, these names are commonly called **identifiers**.Identifiers in Python must not start with a digit and must only use letters (uppercase and/or lowercase), digits and the underscore _. Names must not contain a space.

Some words are Python **key words**, used by the language, and may not be used for anything other than their intended purpose.

Try this exercise:

Which of the following are legal Python identifiers?

upperLimit	upper_limit	upper-limit	upper limit	dollars$	3rdfloor
chapter8	SCORE	air-conditioned		SD&AWSmith	

1.3 Variables and data types

At the most general level of description what most programs do is operate on data. Consequently programs need to do two things: represent / store data, and operate on it. The general term for the parts of a program that store data is **data structures**, and the general term for the description of the processing to be carried out is an **algorithm**.

The most basic data structure is an individual named value. This is referred to as a **constant** if the value cannot or should not change within the program (e.g. maths constants such as pi, fixed values such as the GST rate) or a **variable** if the value can change (e.g. an average, any calculated value). By convention, constants are given identifiers made up of capital letters and underscores. A variable in a programming language is much like a variable in algebra. "Let *x* have the value 2" in a programming language would usually be written something like `x = 2`. For the rest of this chapter we will focus on variables.

A variable is a name/identifier referring to a stored piece of data. The value of the data may be changed while its name/identifier remains the same.

Data types

Individual pieces of data in a Python program can only be of a certain limited range of types. The data types we will be using are:

`int` – integer (number with no fractional part) e.g. 3, 100000000, -25

`float` – a floating point number e.g. 3.065, 10.0, -25.5

A floating point number is a computer representation of a real number. It sometimes has to limit the number of significant digits. See section 2.11.

`str` – a series of characters (letters, digits, punctuation, spaces) which might be a word or a sentence. In programming languages such data is usually called a string - str is short for string. A string is defined within a pair of quotes e.g. "this is a string" or 'this is also a string'

`bool` – has only two possible values, True or False. It is short for Boolean.

Python is weakly typed. Variables in Python do not have a set specified type as is common in many other languages, but get their type from the value that is assigned to them. For example, to create a variable `x` which holds an `int` value of 100:

```
x = 100
```

Later in the program we can simply assign a string value to x, thereby changing the type of value that `x` stores:

```
x = "Hello"
```

Variables that can change their type are very useful and flexible in some circumstances, but they are also a potential source of programming error. This is why many languages (such as Java) use strong types.

An example

The following example will demonstrate the use of different data types.

The specification of the task: store and display some information about an animal – its name, how many legs it has, its height in metres and whether it has a tail.

The algorithm for this task might be

store animal's name data
store animal's leg data
store animal's height data
store animal's tail data
display animal's name data
display animal's leg data
display animal's height data
display animal's tail data

When coded in Python it might look like this:

```
name = "Elephant"
numLegs = 4
height = 4.5
hasTail = True

print(name)
print(numLegs)
print(height)
print(hasTail)
```

The first four statements set up variables which store data of different types.

Type this code into a new window in your IDE. Save it and run it.

This code will produce the output

```
Elephant
4
4.5
True
```

The equal's sign "`=`" is the **assignment** operator. Think of it as "gets the value of". After the statement `name = "Elephant"` is performed, any use of the identifier `name` will produce the string `"Elephant"`.

The second statement declares assigns the value 4 to an integer variable called `numLegs`.

Python data types exercise

Insert four `print type` statements, one after each print statement, as shown below:

```
print(name)
print(type(name))
print(numLegs)
print(type(numLegs))
print(height)
print(type(height))
print(hasTail)
print(type(hasTail))
```

Run the program. The output will inform you of the data type of each variable. The data is assigned automatically to a data type.

Change the assignment of height to 4, run again and note the data type.

Change the assignment of height to 4.0, run again and note the data type.

Write the following lines at the bottom of your program.

```
numLegs = 3.5
print(numLegs)
print(type(numLegs))
```

What is happening to the data type of the variable as you change the value stored in it?

Back to the example

Put the code back to the way it was initially, with just the four assignment statements and four print statements.

Imagine being somebody looking at the output without being able to see the code. It wouldn't make a lot of sense. Anyone looking at it may have no idea what 4, 4.5 or `True` refer to. How the user experiences a program is called the user interface. It is a very important consideration in the programming process.

The code below shows how to add text to label each piece of data.

```
print("Name:", name)
print("Number of legs:", numLegs)
print("Height in metres:", height)
print("Has a tail:", hasTail)
```

Try it. This code should produce:

```
Animal name: Elephant
Number of legs: 4
Height in metres: 4.0
Has a tail: True
```

Comments

Imagine being someone else looking at the code. What is its purpose? Who wrote it? When? This information should be recorded in comments within the code. Comments are notes in the code written for the programmer.

Comment lines in Python are preceded by a #. There is no block comment facility (as there is in many other languages).

A program should begin with the name of the file it is stored in, and a short description of its purpose. Generally it would also record the author and date information e.g.

```
# animal.py
# displays information about an animal
# Joe Bloggs, February 2012
```

Further exercises

Change the statements in the code example to describe

- a chicken
- a spider
- a fish

1.4 About me - a guided walk through the development of a solution.

The task: Write a program to store information about yourself. The information should include your name, your height, your address, your telephone number and whether you have brothers or sisters. A summary of this information should be printed out to the screen.

Where do you start? Our software development process suggests you start with clarifying the problem. It looks straight forward at first, but actually there are many decisions to be made.

Clarify the problem

Is height to be in metres, centimetres or millimetres?

Is address to be stored as one item, or broken up into street number and street?

Can you assume the reader (user) will know which country? Which city?

(What is the context?)

Do you want to store the number of brothers and sisters, or just that there are some?

Do you want to store the information about brothers and sisters together, or individually?

How will you define a brother or sister? Is it someone who is genetically related? Someone who lives in the same house?

How should the data look when it is displayed?

Decide the answers to these questions and any others you come up with.

Design a solution / algorithm

What data needs to be stored?

What names will you use for these variables?

What data type will they be stored as?

Point for discussion : Why might you not want to store your telephone number as an integer?

What order will the information be displayed in?

Use pseudocode or a flowchart to describe your algorithm.

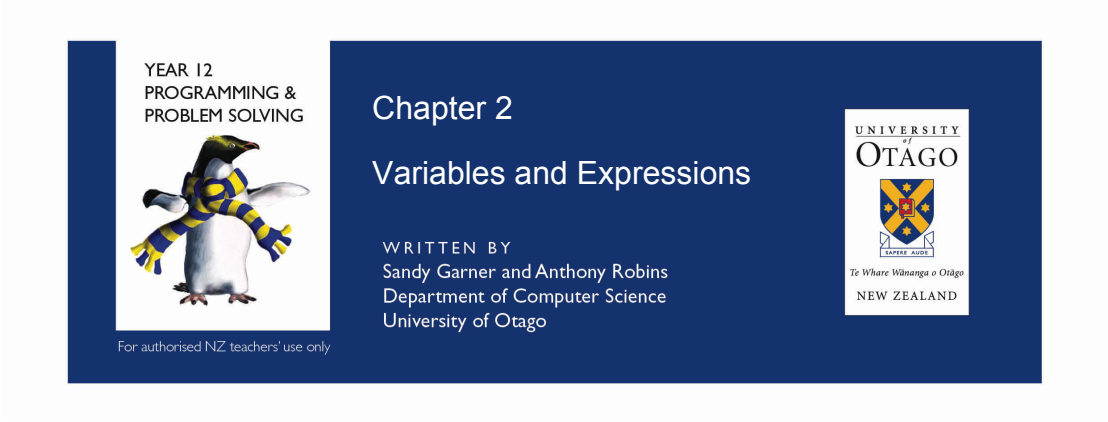
Code the algorithm as a program

Test the program. Once your program is running, check it for accuracy.

Document the program. Imagine being someone else looking at your output. Will they understand it? Is your data labelled clearly? Correctly?

Imagine being someone who is trying to understand your code. Are the variable names (identifiers) sensible and descriptive? Is there a block comment at the top saying what the purpose of the program is? Are any tricky or unusual bits commented?

Think about coming back to your code later, when things may have changed. How easy is it to modify your code to reflect those changes? This will become more important as your programs get more complex.



2.0 Introduction

This chapter further explores the topic of variables in programming languages, and looks at how to code basic arithmetic expressions (simple maths) in Python. We'll also say some more about algorithms, flowcharts and pseudocode, topics that will develop and become much more significant as the example problems and programs that we work on get more complex.

In all the following examples code can be tested by typing it in to a Python program, saving it and running it.

2.1 More about variables and assignment

When a variable is assigned a new value, its previous value is no longer available. After the two statements below are performed in order, y will hold the value 5.

```
y = 3
y = 5
```

What has happened to the 3? The previous value of `y` has been overwritten in memory, like rubbing out a value written in pencil and writing a new one.

The `3` and `5` in this example are called **literal** values. Literal values are values written directly in to the code. Until Chapter 4 (Getting input) all values will either need to be literal or calculated.

A variable can be assigned the value held by another variable.

Code example 1

```
#initial values
lastMonth = "June"
thisMonth = "July"
...
and some time later in the program
...
#roll over to new month
lastMonth = thisMonth
thisMonth = "August"
print("Previous month", lastMonth)
print("Current month", thisMonth)
```

Notice that the text describing each month is written within double quotes, making it a string. What is wrong with this statement:

```
lastMonth = "thisMonth" ?
```

What would the error be if the exact same statements from Code example 1 were performed in the order given below?

Code example 2

```
#initial values
lastMonth = "June"
thisMonth = "July"

#roll over to new month
thisMonth = "August"
lastMonth = thisMonth

print("Previous month", lastMonth)
print("Current month", thisMonth)
```

The order of statements in a program is important.

2.2 Algorithms, flowcharts and pseudocode

Recall that an algorithm is an exact description of a process. Here are the four defining characteristics of an **algorithm**:

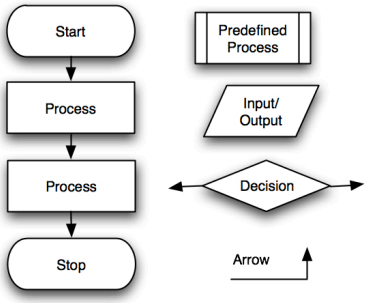
1. An algorithm is described by a sequence of steps.
2. There must be one starting point, and one finishing point must be reached.
3. After each step it must be clear which step is to be performed next.
4. The steps must be composed of basic operations (whose meaning is clear and unambiguous).

In order to solve a given task, specifying the steps in the correct order is essential.

An algorithm can be written in a programming language, as a flowchart, in pseudocode, or in any human language (or other suitable notation). When designing the algorithm, you should always start on paper. Flowcharts and pseudocode are good tools for clarifying the steps required.

A **flowchart** is a useful tool for visually describing the ordered sequence of steps which make up the solution. Flowcharts are particularly useful when dealing with choices and loops (selection and iteration).

A flowchart uses different shaped symbols to represent different kinds of statement. The most commonly used is a plain box, which describes a process. A process may be a single statement or a group of statements. The arrows show the sequence, from Start to Stop.



The right-hand list shows other shapes which will be useful later. Chapter 3 will introduce predefined processes with Modular programming. Chapter 4 deals with user input. Printing text to the display is a form of Output. Chapter 5 deals with selection (choices, decisions). Chapter 6 deals with loops, where the arrow will return to a previous point in the chart, going up rather than the usual down.

A flowchart should have a single Start and at least one Stop point. The sequence, where possible, moves from top to bottom.

Pseudocode is usually some informal English-like description that follows the structural conventions of a programming language. There are no rules for pseudocode. In practice pseudocode is a way of writing down an algorithm which hints at the way it might be coded. Where a flowchart may not suggest which language you are using, pseudocode might use words that belong in your language of choice.

Our preferred style of pseudocode uses numbered instructions. Different levels of numbering can be used to clarify the processes and form them into natural groupings. Indenting the different levels helps provide clarity.

For the month roll-over code in Code example 1, pseudocode might look like this:

```
1. set this month to July
2. rollover to new month
      2.1 set last month to this month
      2.2 set this month to August
```

or this:

```
1. thisMonth = "July"
2. rollover to new month
      2.1 lastMonth = this Month
      2.2 thisMonth = "August"
```

In the latter example, the variable names are already chosen.

Frequently, the comments for your program can come directly from the algorithm or pseudocode. The main parts of the algorithm can be typed in as comments, then the code can be built up under each comment.

2.3 Expressions

A variable can be assigned the value which results from a mathematical expression.

Using operators, values can be combined into expressions that specify other values. For example 2 + 3 is an expression to combine the values 2 and 3, and specify the resulting value 5.

The examples below demonstrate how to write code for simple addition, subtraction and multiplication operations. Each time, the result of the expression is assigned to a variable. **Addition**

```
fingers = 5 + 5
totalEggs = smallEggs + medium Eggs + largeEggs
newPrice = price + 10.50
```

If you wanted to store the `newPrice` back in the `price` variable, you might follow the statement

```
newPrice = price + 10.50
```

with `price = newPrice`

but the same thing can be achieved without the need for the `newPrice` variable at all by simply writing

```
price = price + 10.50
```

This works because the addition operation (+) is performed before the assignment (=). The new value for `price` is calculated **then** stored.

Subtraction

```
difference = 104 - 34.2
temperatureAlert = minimumTemp - 2.5
profit = salePrice - cost
```

Multiplication

```
fingers  = 5 * 2
totalCost = cost * numberOrdered
lengthCubed = length * length * length
```

Some formatting tips:

Say you want to print `$5.00` using an integer value called dollars. You might try

```
dollars = 5
print("$", dollars, ".00")
```

but that will leave a gap between the $ sign and the price.

If you finish a `print` statement with

```
,end=''
```

the cursor doesn't move down to a new line, so now you could achieve `$5.00` by writing:

```
dollars = 5
print("$", end = '')
print (dollars, end='')
print(".00")
```

If the single quotes `''` have a space between them `' '`, the space will be appended to whatever was printed. If the single quotes have a colon between them `':'`, the colon will be appended.

```
winner = "Water Boy"
print("First place", end = ' ') #space between quotes
print(winner)
#prints First place Water Boy

winner = "Water Boy"
print("First place", end = ':')
print(winner)
#prints First place:Water Boy
```

In fact, any sequence of characters between the single quotes will be appended.

```
winner = "Water Boy"
print("First place", end = ': ') #space after colon
print(winner)
#prints First place: Water Boy
```

So you can achieve `$5.00` more concisely by writing:

```
dollars = 5
print("$", end = '')
print (dollars, end='.00')
```

Another way of printing in Python without gaps is not to use the comma, but to build up one long string from two or more smaller strings using the + sign. Each string is appended to the end of the previous string. This is called **concatenation**.

```
# string concatenation example
print ("$" + "5" + ".00") # will print $5.00
```

If any value is not a string, you will need to use the Python string function **str** to represent it as a string:

```
# concatenation with str function example
dollars = 5
print ("$" + str(dollars) + ".00") # will print $5.00
```

2.4 Practical

For each of the following problems design your solution then turn it into a working program. Remember the process:

- Clarify the problem:
  identify the input – the data that makes up the problem, and where it comes from
  identify the output – the data or "actions" that are required and how they should be presented
- Design a solution:
  plan a series of steps which will take you from input to output
  refine this plan until it is completely precise, an algorithm, using pseudocode or a flowchart.
- Code the algorithm as a program
- Test the program.
- If necessary, improve the program, design, code and test again
- Document the program (as you write it)

For all of these problems, don't worry about the formatting of the output of any float values yet. Remember to use descriptive variable names.

Problem 1: Calculate, store and display a team's total game points for 3 matches. They scored 7 in the first match, 5 in the second and 10 in the third.

Problem 2: Your bookseller's giftcard has a starting balance of $50. You buy a book for $35. Display the initial balance, display the purchase information then calculate, store and display the new balance.

```
Initial Card Balance $50.0
Book Purchase $35.0
New Card Balance $15.0
```

Problem 3: Your service station has a stored price list for diesel ($1.47 per litre), 91unleaded ($2.06) and 95octane ($2.17). Each price needs to be increased by 10% and stored again. Display the initial prices, calculate and store the new values, then display the prices again.

```
Initial price:
Diesel price $1.47
95 price $2.17
91 unleaded price $2.06

Price increase of 10%:
Diesel price $1.617
95 price $2.387
91 unleaded price $2.266
```

Problem 4: Convert a height of 1.58 metres to centimetres and print both values.

```
Height in metres: 1.58
Height in cm: 158.0
```

2.5 Constants

A constant should be used wherever a value is fixed and should not be assigned a new value within the program.

Python has no capacity for declaring constants, so we are going to use the programming convention of naming a constant value with upper case letters, and words separated by underscores e.g. `EXCHANGE_RATE`. This will help to remind you that the value is to be used but not altered.

Some constants are predefined by the language.
`math.pi` is a 16 significant decimal place representation of π (but needs the statement `import math` before it can be used.) This predefined Python constant does not follow the convention we have set for ourselves of using uppercase letters.

Problem 5: Your robot with a single-axle design allows you to instruct it how many rotations it should turn the axle which is attached to its driving wheels. If the wheels have a radius of 2.7cm how far will the robot travel with 2 rotations?

The formula pi * diameter will give the circumference of the wheel.

Identify the two constants in this problem.

pi can be rounded to 3.142 :

```
Rotations: 2.0
Distance travelled: 33.9336 cm
```

or you could use the built-in value math.pi :

```
Rotations: 2.0
Distance travelled: 33.929200658769766 cm
```

2.6 Why use variables?

Variables make the code easier to read, more reliable and easier to maintain. Take a look at the three examples below. Each will produce exactly the same output to the screen.

example a

```
print(3, "from High Score")
print(35, "from Low Score")
```

Gives little information to someone looking at the code. We can't see what the numbers mean or how they were calculated.

example b

```
print(38 - 35, "from High Score")
print(35 - 0, "from Low Score")
```

We can't see what the numbers mean but we can see a calculation from which we might be able to work out what is happening.

The calculation is now performed by the computer which makes it more **reliable**.

example c

```
highScore = 38
currentScore = 35
lowScore = 0
print(highScore - currentScore, "from High Score")
print(currentScore - lowScore, "from Low Score")
```

We can see what the numbers represent and it is clear from the names what the calculation is intending to do. This makes the code more **readable.**

It is easier to change the value of `currentScore`, as there is now only one place to do it, and it is clearly labelled. It is also clear where to update `highScore` and `lowScore` should this be required. This makes the code more **flexible** and **easier to maintain**.

2.7 Precedence

As in mathematics, there are clear rules of precedence described in every computer language.

Precedence rules for multiplication, addition and subtraction

Expressions contained within parentheses will be performed first.

Multiplication will be performed before addition and subtraction.

If there are more than one multiplication operations, they will be performed in left to right order.

Addition and subtraction operations have equal precedence and will be performed in left to right order.

Examples

4 + 3 − 5	will evaluate to 2
4 + 3 * 5	will evaluate to 19
(4 + 3) * 5	will evaluate to 35
4 * 3 + 3 * 3	will evaluate to 21
(4 * 3) + (3 * 3)	will evaluate to 21. *It can be helpful to use parentheses to add clarity to a calculation.*
4 * (3 + 3) * 3	will evaluate to 72
5 − 2 * 2 + 3	will evaluate to 4

2.8 Practical

For each of the following problems identify the constants and variables required, design your solution then turn it into a working program. Keep in mind issues of readability, reliability and maintainability.

We have given suggestions for what your output might look like. Generally the output will offer enough information for your answer to be checked by someone looking at the screen.

Problem 6:

Calculate, store and display the total price of a dinner for two, given the price of a main course is $12.50, the price of a dessert is $6 and the price of a drink is $3.55.

Make sure your program is flexible enough to cope easily with a different number of mains, desserts or drinks.

```
2 mains at 12.5 is 25.0
2 desserts at 6.0 is 12.0
5 drinks at 3.55 is 17.75
Price of meal: $54.75
```

Problem 7:

Your iTunes account has a starting balance of $100. You buy 3 short tracks at $1.79, four medium tracks at $2.50 and 2 long tracks at $4.50.

Calculate, store and display your new account balance.

```
Initial balance: $100.0
Purchased 3 short tracks at 1.79
Purchased 4 medium tracks at 2.5
Purchased 2 long tracks at 4.5
Final balance: $75.63
```

Problem 8:

I have 2 bank balances, one contains $100.00 in US$, the other $100.00 in AUS$. Write a money conversion to display your total worth in NZ$.

The exchange rates at the time of writing were US 1.154 and AUD 1.22

```
US$100.0 exchanges to NZ$115.39999999999999
AUD$100.0 exchanges to NZ$122.0
NZ balance: $ 237.39999999999998
```

Problem 9:

A rugby team has scored over the season 55 tries, 35 conversions, 48 penalties and 5 drop kicks. The points are 5 for a try, 3 for a penalty and a drop kick, 2 for a conversion. Calculate and display the total points for each category, and for the season.

```
Season try points(5): 275
Season drop kick points(3): 15
Season conversion points(3): 70
Season penalty points(2): 144
Total season points: 504
```

Problem 10:

A family has spent the afternoon picking berries, and will be charged by the total weight at the rate of $5 per kg. They have picked 9 buckets, 13 boxes and 14 punnets.

A punnet weighs 250grams. A box weighs 1.5kg. A bucket weighs 7 kg.

Calculate and display the total charge.

```
Total punnet weight: 3.5
Total box weight: 19.5
Total bucket weight: 63.0
Total berry weight: 86.0 kg
Price to charge: $430.0
```

2.9 Division

The examples below demonstrate how to perform division.

Division

```
speed = distance / time

fairShare = totalCost / numberPersons

averageTemp = (temp1 + temp2 + temp3) / 3
```

Precedence rules for multiplication, division, addition and subtraction

Division and multiplication operations have equal precedence. If there are more than one division and/or multiplication operation in a statement they will be performed in left to right order.

Division and multiplication will be performed before addition and subtraction.

Expressions contained within parentheses will be performed first.

2.10 Practical

For each of these problems, design and code a solution which produces the output shown for the values suggested.

Problem 11a:

I have $66.00 to buy lunch for 12 people. How much can each person spend?

```
$66.0 for 12 people
Each person can spend: $ 5.5
```

Problem 11b:

I have $67.00 to buy lunch for 12 people. How much can each person spend?

```
$67.0 for 12 people
Each person can spend: $ 5.583333333333333
```

Problem 12:

Your robot (as described in Problem 5) allows you to instruct it how many rotations it should travel. If the robot's wheel has a radius of 2.7cm how many rotations should you instruct it to perform in order to travel exactly 2 metres?

pi can be rounded to 3.142
the formula pi * diameter will give the circumference of the wheel

```
Rotations required to travel 200 cm: 11.787726619044252
```

Each of these last two problems uses at least one floating point number. A floating point number will be displayed with at least one decimal place even if it is a whole number e.g. `3.0`. We can tell that the travel distance in Problem 12 was stored as an

integer by the way it is displayed in the output. It is often important to know whether you are dealing with an integer or a floating point number. It is also important to know how to convert data permanently or temporarily from one to the other, and the implications of this conversion (see section 2.11).

2.11 Floating point numbers

Floating point numbers are just an approximate representation of real numbers. Take for example one third, which is exactly represented by the fraction 1/3 but cannot be represented accurately on a calculator or a computer. There has to be a limit placed on the number of digits after the decimal place as there is no natural end point. 0.333333333....

In Python, the data type `float` represents real numbers approximately as floating point numbers.

Floating point numbers are a compromise between the storage space required for a number's significant digits and precision. It works in the same way as scientific notation, where the most significant digits (usually up to15) and the position of the decimal point relative to them are stored. Sometimes, you will see the result of a floating point calculation showing a number such as 5.30000000000000004 where you might have been expecting 5.3.

Try this

```
i = 1.1001
print (i + i + i)
```

Problems like this never happen when using integers. **How do you know if you are using a floating point number or an integer?** As long as one of the numbers being operated on is a float, the result of the expression will be a float, except for division.

The result of the expression `3 + 2` is of data type `int`
The result of the expression `3 + 2.0` is of data type `float`

The result of the expression `3 * 2` is of data type `int`
The result of the expression `3 * 2.0` is of data type `float`

The result of the expression `3 / 2` is of data type `float`
The result of the expression `3 / 2.0` is of data type `float`

Python has an integer division operator which looks like this `//` and is discussed in section 2.12

The result of the expression `3 // 2` is of data type int
The result of the expression `3 // 2.0` is of data type float

As the next section will explain, integer division has its own unique set of behaviours.

2.12 Integer division

While Java and many other programming languages use integer division bydefault, recent versions of Python use floating point division by default, but can perform integer division using the `//` symbol. The core concept of integer division is that when an integer is divided by an integer, the result is also an integer. Any fractional part is discarded e.g.

```
wholeDozens = numEggs // 12
#if numEggs is 27, wholeDozens would be 2
```

Some examples of integer division behave as you would expect e.g.

`6 // 1` evaluates to 6
`6 // 3` evaluates to 2
`6 // 2` evaluates to 3

but here is integer division making a difference:

`6 // 4` evaluates to 1
`6 // 5` evaluates to 1
`6 // 6` evaluates to 1

`1 // 6` evaluates to 0
`2 // 6` evaluates to 0
`3 // 6` evaluates to 0 etc.

The result only takes the whole number part of the calculation. It does not round, as you might expect in mathematics. It truncates – chops off – the fractional part.

A `float` can easily be converted into an integer:

```
temperature = 15.0
temp = int(temperature) #this is the conversion statement
print("Temperature is ", temp, "degrees")
```

This will give you a nice neat display: Temperature is 15 degrees.

But beware of the data loss. Let's change this example slightly:

```
temperature = 15.9
temp = int(temperature) #this is the conversion statement
print("Temperature is ", temp, "degrees")
```

This will also give you the nice neat display: Temperature is 15 degrees. But the temperature has lost 0.9 degrees. It was not rounded, but truncated.

An integer can also be converted to a float e.g.

```
temp = 15
tempReal = float(temp)
```

2.13 Modulo / Remainder

The remainder from integer division can be captured using the modulo (often shortened to mod) operator, which is written using the % sign.

```
5 // 2 evaluates to 2
5 % 2 evaluates to 1          (because 2 * 2 + 1 = 5)

15 // 7 evaluates to 2
15 % 7 evaluates to 1         (because 7 * 2 + 1 = 15)

15 // 3 evaluates to 5
15 % 3 evaluates to 0         (because 3 * 5 + 0 = 15)
```

Try this exercise:

A band of 14 buskers has a pile of 161 dollar coins to share out evenly. Write expressions to determine how many coins they will have to turn into smaller change, and how much each busker should get in total.

2.14 Practical

Problem 13:

You have 18 bars of chocolate and want to divide them evenly among 5 people. It is possible to break each bar into 7 squares. How many whole bars of chocolate will each person get? How many extra squares of chocolate will each person get? How many squares will be leftover?

```
Whole chocolate bars each: 3
Extra squares each: 4
Squares left over: 1
```

Problem 14:

There are a fleet of buses available which can take 38 passengers each. There are 613 children going on a school trip. How many buses will be needed? How many children on average should be on each bus?

```
Buses needed: 17
Average children per bus: 36.05882352941177
```

Problem 15:

There are 131 children who want to play netball. Each team needs 7 players and 2 subs. How many teams will there be? How many children will not be in a team without finding more players?

```
Number of teams: 14
Number of children left over: 5
```

Problem 16:

Given a number of seconds, display the length of time in terms of whole minutes and seconds.

```
184 seconds is 3 minutes and 4 seconds
```

Problem 16 extension:

Given a number of seconds, display the length of time in terms of hours, minutes and seconds.

```
3762 seconds is 1 hours 2 minutes and 42 seconds
```

Problem 17:

The photocopier has jammed. It has printed 105 sheets of a job that was printing 5 copies of a 30 page book. How many books have been safely printed? How many pages of the unfinished book were printed before it jammed?

```
Complete books printed: 3
Unfinished book pages printed: 15
```

Problem 18:

I have $20. I want to buy a chocolate bar (2.50) for each of my 3 friends, and spend the change on apples (50c). How many apples will I get?

Make this program flexible (easy to modify) because next week I may have more friends, or the price of chocolate or apples may have changed.

```
Money for chocolate: $7.5
Money for apples: $12.5
Number of apples: 25
```

2.15 Some more challenging problems.

Don't forget to design the solutions on paper before you code.

Problem 19:

You have $50. You make a purchase of $22.69. What is the change?

Determine what denomination of notes/coins the change should be given in.

```
Purchase : $22.69
Change from $50.0 :
$20 notes: 1
$10 notes: 0
$5 notes: 1
$2 coins: 1
$1 coins: 0
50c coins: 0
20c coins: 1
10c coins: 1
```

Problem 20:

A bus leaves at 8:15am and arrives at its destination 80 minutes later. Given the hour and minute of this (or any other) departure, calculate and display the time of its arrival (assuming it arrives on the same calendar day, and using a 24 hour clock).

```
Departure time 8:15
Journey length is 80 minutes
Arrival time 9:35
```

Problem 21:

You run an event management business. Your sporting venue which seats 3000 people is full. All of these people need to be transported away to lunch after the event. There are 95 taxis in town, which take 4 people, all of which you have booked. There are 45 buses, which seat 40, waiting. How many people will be left waiting after all the transport has left the venue with a full load? How many buses should be asked to make a second trip?

```
People left waiting: 820
Buses to return for second trip: 21
```

Problem 22: The tempo marking on a piece of music reads ♩=84, which means there are 84 crotchet beats per minute. If the music is in waltz time (3 crotchet beats per bar) how many bars will be in a 4 minute piece of music?

```
Number of bars 112
```

Problem 23: A knitting pattern for a jersey says it will take 16 skeins of wool. You want to knit it with stripes which use colours in the ratio dark blue 4: light blue 3: yellow 2: red 1. Store the information, calculate and display how many skeins of each colour will you need to buy?

```
Dark blue 7
Light blue 5
Yellow 4
Red 2
```

2.16 End notes

Assignment shortcuts

`a += b` is the same as `a = a + b`

`a -= b` is the same as `a = a - b`

`a /= b` is the same as `a = a / b`

`a *= b` is the same as `a = a * b`

Divide by zero error

Be careful with any expression which might end up with a division by zero. This will produce a runtime error.

Scientific notation / E notation

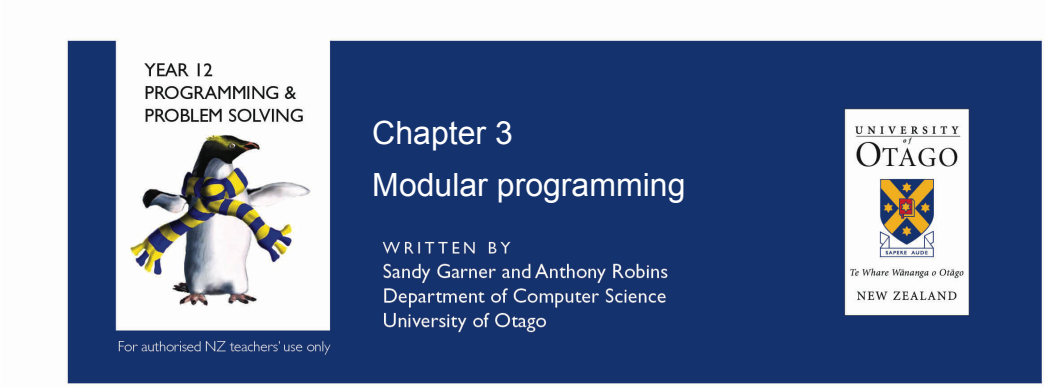
Very large numbers and very small numbers are often represented by scientific notation. In computing and on calculators, the "times 10 to the power" of is generally represented by the letter e.

Decimal notation	Scientific notation	Interpretation
356	3.56e2	3.56 * 10 to the power of 2
123450.0	1.2345e5	1.2345 * 10 to the power of 5
0.0032345	3.2345e-3	3.2345 * 10 to the power of -3

2.17 Revision - Concept list from Chapter 1

How many of these can you remember?

algorithm	identifier
pseudocode	block
text	integer
variables	data
data type	string
real	floating point
boolean value	interpreter
statement	execute
IDE	comments
file	parentheses
case sensitive	key word
assignment	output
user interface	float



3.0 Introduction

Module is a general term for the structures which in specific programming languages are called procedures, functions, methods or subroutines. In the text we will often use the general term **module**, though specificallyin Python we are referring to **functions**.

Two ways of viewing what is going on in a program help us understand and describe its behaviour. "Flow of data" refers to the journey that data might make through a program i.e. the way it is input, organised, processed, transformed, and finally, output. "Flow of control" refers to the sequence in which the parts of a program execute and interact (at its most basic level the order in which the statements of the program execute).

Modularisation is a core tool for organising flow of control. Modules are much more useful if we can send data to and get data back from them, a way of managing the internal flow of data.

3.1 Modular programming

The simplest Python program consists of just a **main routine** - a sequence of un-indented statements When written in a file these statements are described as **file level** statements. All of our example programs so far have had this simple structure. The statements are executed in order, then the program finishes.

When a program is more than a few statements long, however, its readability and maintainability is usually improved if statements which belong together to perform distinct portions of the task are grouped together into named functions. This the best way of solving a complex problem: breaking it down into smaller pieces. **Calling** a functions(executing / performing the statements that it contains) requires just a single statement based on the function's name.

Modular programming is also convenient for sequences of code which need to be executed repeatedly. Rather than write the whole sequence of statements out again and again, the statements can be written just once in a function, and the function can be called repeatedly.

Some thought should go into choosing an appropriate name (identifier) for the function. A name should be descriptive but not too specific e.g. if a piece of code will be used to average test scores, but might also be used to average the time taken, then `average` would be a better name for the function than `averageScore`.

A function has a **definition** - a statement starting with the keyword **def**, followed by the function's name, a set of parentheses `()` (which may or may not be empty) and a colon `:` . This statement is sometimes called the header. The statements which describe the behaviour of the function follow, each indented the same number of spaces (usually 4). This indentation is not optional. Indentation is the way that Python determines the structure of functions (and as we shall see later also other language constructs).

Function example:

```
# a function should start with a comment briefly describing its purpose
def functionOne():
    #the function's statements go here, indented to the same level, usually 4 spaces
    print("line1")
    print("line2")
```

The indented statements form the **body** of the function. The function definition is finished when the indenting stops.

IDLE does this kind of indentation automatically. Use the backspace key to break out of indentation.

Any function must be defined / declared **before** it is called. Hence Python programs usually have function definitions listed first, and the `main` routine last. Functions are called by using a statement that consists of their name, including the parentheses. Here is a complete program with the same function definition as before, but now with a `main` routine which calls the function:

```
#function definition
def functionOne():
   print("line1")
   print("line2")

#main routine calls functionOne two times
functionOne() #this is a call to the function
functionOne() #this is a second call to the function
```

If a function is never called, it will never be executed.

A function can be called from within another function.

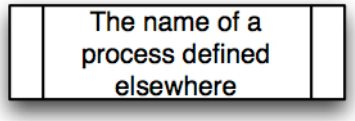
What do you think the output of the function example program will be? Try it out and see if you are right.

Program design using modules

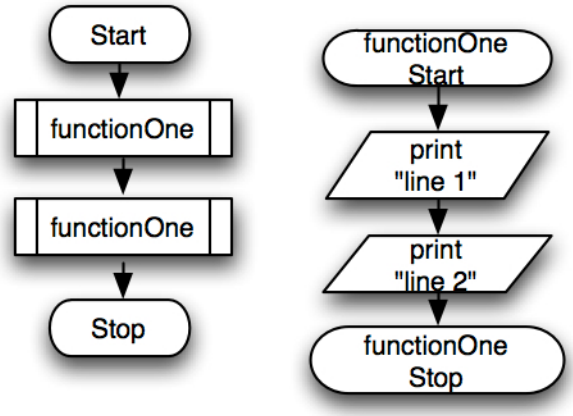
When designing a solution, it isn't necessary straight away to know how you might implement each and every module. Just having a name describing what a module should do is a good starting point. When you are happy with the overall structure of the solution, then you can design the inner workings of each module.

Modularity and flowcharts

Now that programs are modular, a simple sequence no longer works. When drawing a flowchart, each function should be represented in the main flow as a box with an extra line on each side. Then each function should have its own individual flowchart.



The example on the previous page could be represented like this:



Modularity and pseudocode

The program example we have been using can be described in pseudocode. As with the flowchart, the main structure of the program is listed first, then the pseudocode for any functions follows.

```
main flow
1. CALL functionOne()
2. CALL functionOne ()


functionOne
1. print "line 1"
2. print "line 2"
```

3.2 Scope

Global variables

If you want a variable to be available to every function it should be created in the `main` routine. Such a variable is called a global variable. All the variables you have created so far have been global variables.

Local variables

Any variable created inside a function is considered local to that function. A local variable cannot be used outside of the function unless it is specifically declared as global.

Scope

Local variables and global variables would usually have different names, but it is possible for them to have the same name. This raises for the first time the concept of scope, which is about where a variable is accessible within the structure of the program.

In the following example the name "`x`" belongs to a local variable in `functionOne` and the global variable in `functionTwo` and the main routine.

```
#function definitions
def functionOne():
    x = 8               # a new local variable x is created
    print("x is", x)    # accesses the local variable x:  prints x is 8

def functionTwo():
    print("x is", x)    # accesses the global variable x: prints x is 5

# main routine
x = 5                   # a new global variable x is created
print("x is", x)        # accesses the global variable x: prints x is 5
functionOne()
functionTwo()
```

Note that in `functionOne` the assignment `x = 8` is interpreted as creating a new local variable called `x` (rather than using the global `x` from the `main` routine). In order for a function to assign a value to a global variable the variable must be declared as global within the function. Ideally such global variables are listed at the beginning of the function.

```
#function example, assigns a value in the function to an existing variable
def functionOne():
    global x     #  the x created in the main routine is now global
    x = 8        #  assigns a new value to the global variable x

#main routine
x = 5                   # a new global variable x is created
functionOne()
print("x is ", x)       # prints x is 8
```

Note that if no global variable `x` was declared in the `main` routine then the global declaration in `functionOne` would create one – accessible in other functions and the `main` routine just as if it was declared in the main routine itself. (For example in the program above if the statement x = 5 did not exist the behaviour of the program would not change.) It is good practice, however, to declare all variables intended to be global at the start of the `main` routine.

Copy these example programs into your IDE and experiment with them!

3.3 Practical

Problem 1:
The traditional ballad Clementine has a verse chorus verse chorus structure. Write a program in which each verse is written by its own module, as is the chorus. Call the verses and the chorus in the right order to print out the lyrics in full (including every chorus).

In a cavern, in a canyon,
Excavating for a mine
Dwelt a miner forty niner,
And his daughter Clementine

Chorus:

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine

Light she was and like a fairy,
And her shoes were number nine,
Herring boxes, without topses,
Sandals were for Clementine. -- **Chorus**.

Drove she ducklings to the water
Ev'ry morning just at nine,
Hit her foot against a splinter,
Fell into the foaming brine. -- **Chorus**.

Problem 2:
Your program has six stored integer values: **6, 17, 2, 15, 0** and **3**

Write 2 functions in your program. One function should calculate and display the average of the values, the other should calculate and display the sum of the values. Call these functions from the `main` routine.

3.4 Getting data in to a module

The real power of computing starts to be unleashed when we are able to apply a module repeatedly to different data. The data might be stored somewhere that the module can access it such as afunction accessing global variables. Usually, however, it is better design to send the input data directly to the module in the form of input or inputs specified in its parentheses. These inputs are usually called **parameters** or **arguments**.

This works in the same way as the sine function on a calculator "sin(x)". The sine function must be sent an input to work with. We have been sending inputs to a function right from our very first program. Specifically we have been sending the data that we want to print out to the`print()` function e.g. `print("Hello")`.

The function definition must create in its parentheses `( )` a local variable for each input it is being set up to receive. These input parameters operate as local variables, only accessible within thefunction body.

```
def functionTwo(s): # set up for one input, names it s
    print(s)
    #s is a local variable, able to be used only within this function
```

In the main routine this function could be called as follows:

```
functionTwo("Hello")
```

In the next example, the function uses its parameter value in a calculation.

```
def functionThree(i): # set up for one input, names it i
   threeMore = i + 3
   print(threeMore)
   # i and threeMore are local variables, able to be used only within this function
```

In the main routine this function could be called as follows:

```
functionThree(39)
```

The variables that afunction is set up to receive as input are called the **formal parameters** of thefunction. The corresponding values sent when thefunction is called are called the **actual parameters**.

More than one input can be sent to a module as long as the numberof the actual parameters matches the formal parameter declaration(s). In short the call to the module must provide the "expected" values.

```
def functionFour(i, d):
    # set up for two inputs, names the first i & second d
    print("Sum is",(i + d))
    # i and d are local variables, able to be used only within this function
```

In the `main` routine this function could be called as follows:

```
functionFour(2, 3.14159)
```

The calls below are for the functions defined above, and could be used in the `main` routine or other function in the same program.

```
functionTwo("the rain in Spain")
functionTwo("Print me")
functionThree(67)
functionFour(4, 5.7)
functionThree(4 + 3)
functionFour(4.5 * 2.3, 5)
i = 45
functionThree(i)
```

What happens if you use these function calls?

```
functionThree("PrintMe")
functionFour(3)
functionTwo()
```

What happens if you use this function call?

```
functionFour("Hi","There")
```

Because Python is a weakly typed language, `functionFour` will obligingly add integer parameters and concatenate string parameters. This may produce results the programmer did not intend!

3.5 Practical

Problem 3:
You want to send the same letter to 5 different people, addressed individually. Plan and write a modular program which can do this. The name of the person to whom the letter is addressed should be the input variable to a module which is called 5 times.

```
Dear John Smith
This is to inform you that you have been selected . . .

Dear Joe Bloggs
This is to inform you that you have been selected . . .

Dear Jane Doe
This is to inform you that you have been selected . . .

Dear Mary Major
This is to inform you that you have been selected . . .

Dear Richard Roe
This is to inform you that you have been selected . . .
```

Problem 4:
Two verses of the Beatle's song *Blackbird* are shown below. They differ only a little - mark these places. Write a module which will take enough inputs to receive the text for each of the differing sections. In the module, write code to print out the verse (including the input data). Call this module twice, once for each verse, sending the correct information to print the verses as shown.

> Blackbird singing in the dead of night
> Take these broken wings and learn to fly
> All your life
> You were only waiting for this moment to arise
>
> Blackbird singing in the dead of night
> Take these sunken eyes and learn to see
> All your life
> You were only waiting for this moment to be free

Problem 5:
Your `main` routine has a stored value for a purchase which does not include GST. Write a module which takes the purchase price as an input parameter and prints out the price excluding GST, the GST amount, then the price including GST. Make sure your displayed information has clear descriptions.
Call your module once to display the data for your purchase.
Adapt your program to display the values for a second purchase. How easy is it?

```
Price excluding GST: $ 35.6
          GST added: $ 5.34
Price including GST: $ 40.94
```

Problem 6:
A train is scheduled to leave Wellington for Johnsonville and all stations in-between at 3.02, 3.28 and 3.40. Each train takes 7 minutes to travel to Crofton Downs, 2 minutes to travel to Ngaio, 2 minutes to Awarua Street, 2 minutes to Simla Crescent, 1 minute to Box Hill, 2 minutes to Kandallah, 3 minutes to Raroa and 2 minutes to Johnsonville.

Plan and write a modular program which will produce the output below (on 3 long lines).

Wellington 3:02 Crofton Downs 3:09 Ngaio 3:11 Awarua St 3:13 Simla Cres. 3:15 Box Hill 3:16
Kandallah 3:18 Raroa 3:21 Johnsonville 3:23

Wellington 3:28 Crofton Downs 3:35 Ngaio 3:37 Awarua St 3:39 Simla Cres. 3:41 Box Hill 3:42
Kandallah 3:44 Raroa 3:47 Johnsonville 3:49

Wellington 3:40 Crofton Downs 3:47 Ngaio 3:49 Awarua St 3:51 Simla Cres. 3:53 Box Hill 3:54
Kandallah 3:56 Raroa 3:59 Johnsonville 4:01

3.6 Getting data out of a module

The modules you have been using so far have been called for the effect produced by executing their statements. A module can also be used to calculate a value and **return** that value to where it was called. This is similar to the sin(x) function on a calculator returning a result (the sine of the input x).

Functions in Python can return a specific value by ending with the keyword `return` followed by a value. To use these return functions just use the function call in your program where the returned value is required. For example:

```
# function returns an integer value
def functionFiveA():
    x = 5
    return x
# main routine
print( functionFiveA() )   # prints 5
```

All functions in Python return a value. If no value is returned explicitly then the value returned automatically is `None`.

```
# function returns nothing
def functionFiveB():
    x = 5
# main routine
print( functionFiveB() )   # prints None
```

The value returned by a function may or may not be assigned to a variable, depending on the circumstances. If the value is to be used in other calculations, or used more than once, it should probably be stored. This is particularly important if any of the values involved may have changed. If the value is only to be displayed, or used in just one calculation, the function should probably just be called at the point that the value is required.

For example assume that a program has the following function:

```
#function takes 2 input parameters and returns a single value
def functionSix(i, d):
    return i + d
```

Using this main routine the returned value is stored, then printed:

```
# main routine - sum is assigned the value returned by the function
sum = functionSix(4, 5.7)
print(sum)
```

Whereas using this main routine the returned value is used (printed) right away:

```
# main routine - the value returned by the function is displayed but not stored
print(functionSix(4, 5.7))
```

Naming

The name of the function can be a huge aid to the program's readability. Using the word **get** as a prefix to the name of a return function is common practice e.g. getCost, getTotal, getAverage. If get is not appropriate, some other verb might be used e.g. calculate, add . . .

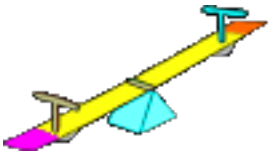
The readability of `functionSix` would be improved if the name of the function was more descriptive. Here it is again with a more appropriate name:

```
def getSum(i, d):
    return i + d

#main routine
print(getSum(4, 5.7))
```

3.7 Practical

Problem 7:

To balance a see-saw, the mass * distance on one side of the fulcrum needs to balance the mass * distance on the other side. A 46 kg child is sitting 2 metres from the fulcrum. Write a modular program which will calculate the distance from the fulcrum a second person needs to sit to balance the see-saw. (The weight of the second person will be the input parameter to the module.)

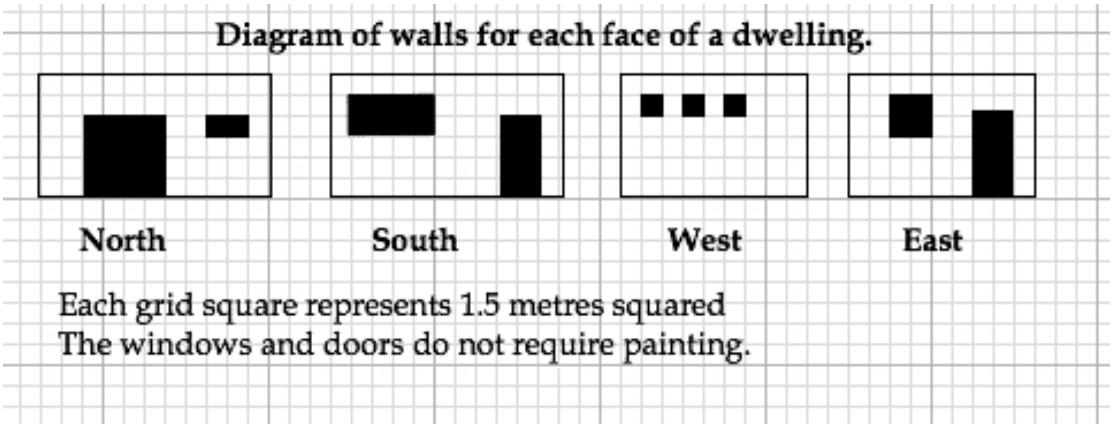


Distance for 75 kg person: 122.66666666666667 cm
Distance for 60 kg person: 153.33333333333334 cm
Distance for 100 kg person: 92.0 cm

What would be a good test weight to use to check your calculations?

Problem 8:

You have been asked to calculate how many litres of paint will be required to paint a house. Write a modular program which calculates the paintable area of each outside wall of a house separately then displays the litres of paint required. One litre of paint will cover 16 square metres. Two coats will be needed. How much paint needs to be ordered according to the dimensions in the diagram?



Total paintable area: 429.75 square metres

Paint required for 2 coats: 53.71875 litres

Problem 9:

Imagine a rope that fits snugly all the way around the Earth like a ring on a person's finger. Now imagine the rope is made just one metre longer and lifted uniformly off the surface until it is once again taut. What will its height be above the surface?
The answer is remarkable: about 16 cm. It comes simply from the formula for the circumference of a circle. If the extra radius of the rope is r and the Earth's radius is R,

then $2\pi(R + r) = 2\pi R + 100$ so that $r = 100/2\pi = \sim 15.9$

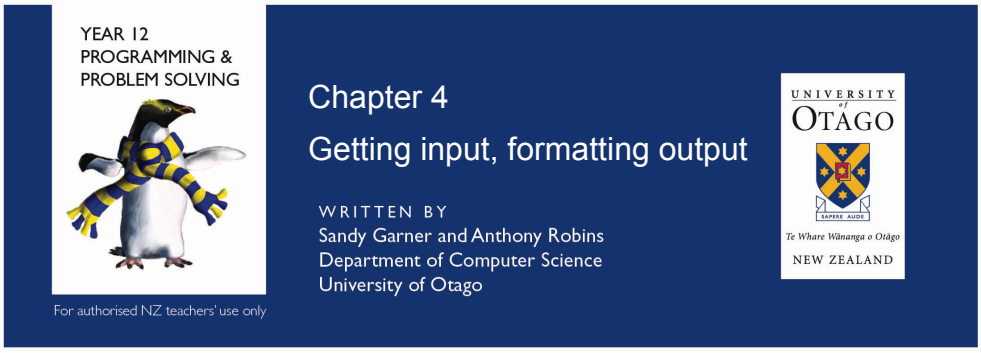
(This version of the problem and solution is from `http://www.daviddarling.info/index.html`)

We don't actually need to know the actual radius or circumference of the earth. So it stands to reason that the same result (about 15.9 cm) will occur for a circle of any size compared to another with a circumference 1 metre (100cm) longer.

Write a module which takes the circumference of a circle as input and returns the circle's radius. Call this module from the main routine in order to show the difference between radii of a selection of circles of different circumference and corresponding circles 1 metre (100 cm) longer.

Write another module which shows you the radius of each of the smaller circle pairs.

```
First circle pair: 1cm 101cm
Smaller circle's radius: 0.15915494309189535
Larger circle's radius: 16.07464925228143
First circle pair perimeters are 15.915494309189533 cm apart.

Second circle pair: 100cm 200cm
Smaller circle's radius: 15.915494309189533
Larger circle's radius: 31.830988618379067
Second circle pair perimeters are 15.915494309189533 cm apart.

Third circle pair: 500cm 600cm
Smaller circle's radius: 79.57747154594767
Larger circle's radius: 95.4929658551372
Third circle pair perimeters are 15.915494309189526 cm apart.

Fourth circle pair: 63700000cm 63700100cm
Smaller circle's radius: 1.0138169874953734E7
Larger circle's radius: 1.0138185790448042E7
Fourth circle pair perimeters are 15.915494307875633 cm apart.
```



4.0 Introduction

The purpose of this chapter is to show how to get information from the user via the keyboard, and how to format numerical output for display or storage. Because we are dealing with an object oriented language, by necessity some objects will be required to achieve this purpose. An understanding of object-oriented programming is beyond the scope of this workbook. If you can receive user input and format output by following the patterns provided in this chapter, it is not necessary that a deep understanding of objects be gained at this stage.

Most languages come with libraries of pre-written code to perform various common or useful tasks. In this chapter we will be using classes and functions from the libraries, so we need to start with a bit of background about what these are.

In Python the standard libraries contain useful classes, methods and other structures. To explain what these are we need to define some key terms. In Python everything that can be given a name is an **object**. For example when we say x = 3 we give the value 3 a name. Thus an integer is an object. So is a string e.g. `s = "Hello"`.

In Python every object is made from a **class**. So far we have talked about values (like 3) having a type (like int), but types and classes are the same thing. To say that 3 is of type `int` means that it is an object made from the class `int`.

A **method** is a function which is part of a class. We call a method by specifying the value / object to call it on, followed by a ".", followed by the method name, for example:

```
"My String".lower()
```

"My String" is one way of specifying an object made from the class `str`, so this statement is calling the `lower()` method on the "My String" object, and will return 'my string'. This makes a copy of the string in lower case (this is covered in chapter 8).

```
n.__add__(4)
```

Given `n = 6` (an object made from the class `int`) then the statement above will call the `__add__()` method on the object n, which will return 10. This is another way of doing addition. (The __ symbol is 2 underscore characters.)

In this chapter we will be using the `format` method of the Python `str` class.

4.1 Input from the keyboard

Obtaining input from the keyboard is relatively simple in Python. By default the input is a string value. It must be cast / converted into any other data type. The example below shows two cases each of a string, an int and a float being stored after being typed in on the keyboard.

```
uni = input("Name your nearest university ")
bikeCount = int(input("How many bicycles do you own? "))
airFare = float(input("Price of Dunedin/Auckland airfare? $"))
name = input("What is your name? ")
birthYear = int(input("What year were you born? "))
halfAge = float(input("What is half of your age? "))
```

Note that `halfAge` may not be an integer, so the code needs to be set up to receive a float rather than an integer.

Type the statements above into a file and run the code. When you run the program it should be quite obvious to you as a user what kinds of values you should type. It may be less obvious that you should press return/enter to finish your input. The user may not be sure whether to enter the dollar sign for a price.

The prompt to the user is an essential part of the process of getting user input. Imagine being presented with an empty space and no prompt to say what should go in it. Taking care to make the user's experience comfortable and sensible is a very important part of program design.

4.2 Practical

In the example outputs below, the characters typed by the user have a box around them.

Problem 1:

Write a program which asks you what value giftcard you have been given, asks you for the name of the book you bought, how much it cost then displays a summary of what you have bought and how much you have left on your card.

```
What is your initial card balance? $ 100
What is the name of the book purchased? Pride and Prejudice
How much did Pride and Prejudice cost? $ 29.95
Initial card balance $ 100.0
Purchased Pride and Prejudice at $ 29.95
New card balance $ 70.05
```

Problem 2:

Write a modular program which takes as input from the user the weight of flour in grams and displays the equivalent number of cups using the formula 125g flour = 1 cup

```
How many grams of flour? 350
Equivalent cups: 2.8
```

Problem 3:

Write a modular program which takes as input from the user their height in metres and displays the equivalent height in whole inches, and in feet and inches. Use the formula 1 metre = 39.37 inches.

```
What is your height in metres? 1.7
1.7 metres is equivalent to 66 inches
or 5 feet and 6 inches
```

Problem 4:

Adapt the GST program you wrote in Chapter 3 Problem 5 so it takes the dollar value of a purchase as an input from the user (and as before displays the amount of GST to be added and the price including GST).

```
What is the pre GST price? 10.00
Price excluding GST: $ 10.0
          GST added: $ 1.5
Price including GST: $ 11.5
```

4.3 Error handling

Unfortunately, once you allow a user to interact with your program you lose a certain amount of control. Programs have to be able to cope with not only good data, but also badly formatted or unexpected data.

Casting the input to `int` will fail if, for instance, the user types **eight** instead of **8**. The text "eight" can not be converted into an integer so it will cause a "run time error" - the code will run until the error occurs, then it will stop and produce an error message. Or the user may simply have typed the wrong key accidentally, or pressed the return/enter key too soon. For a program to run reliably, the possibility of errors like this must be dealt with.

One way of minimising errors is to be careful to tell your user exactly what they should type. If they might want to use a decimal point, either allow for that in your coding or instruct them not to. It might be useful in some circumstances to give an example of the expected input format.

No amount of careful explaining is going to stop the user occasionally pressing the wrong key by mistake. In order that this sort of error does not cause the program to crash, a try except block can be used.

Error handling with try except

The code below has extra statements which will cope with unintended input. If there is an error when the code in the `try` block is executed, the code in the `except` block will be executed instead, then (in either case) the program will continue on to the next statement after the `except` block. *#reads an int from the keyboard*

```
userInput = 0
try:
    userInput = int(input("Choose a number "))
except ValueError:
    print("Not a number")
print("Choice is",userInput)
```

What will be displayed if the user does not type an integer?

Why is `userInput` given a value before the `try except` block?

4.4 Formatting numerical output

Formatting is not something that happens to a number value itself, but something which happens to the string representation of the number as it is being printed.

We will provide you with several examples which should help you get comfortable with output formatting in Python.

The default print format shows float values to 16 decimal places. Python uses the letter **g** for "general format". General format by default limits the output to 6 digits, plus the decimal point. If the value is large, it will be formatted using scientific notation. If there are many digits after the decimal place, the last place will be rounded. The syntax will feel awkward at first, but you will soon learn to use the patterns required. Here are some examples of general formatting. The quotes may be single or double quotes.

```
# rounds and limits number of digits
print('{0:g}'.format(3.144567898765443))

third = 1/3  #  these variables will be used in the examples below
twoThirds = 2/3

print(third)    # the default format is to show 16 places after the decimal point
# in these examples the 0 is optional
print('{0:g}'.format(third)) # g format, rounds, limits number of digits
print('{0:g}'.format(twoThirds))
```

Python uses the letter **f** for "floating point format". The f requires a decimal point and a digit preceding it to specify how many places are required after the decimal point (or it will default to 6).

```
print('{0:.3f}'.format(third))  # rounds to 3 decimal places after point
print('{0:.3f}'.format(twoThirds))
```

One important difference between f and g formatting is that g drops all the trailing zeroes (those to the right of the last significant digit)

```
print('{0:f}'.format(3.140))  #forces 6 digits after decimal point
print('{0:g}'.format(3.140))  #drops all 0's to the right
```

Type the formatting statements above in to your file and record the output.

```
'{0:g}'.format(3.144567898765443)
```
_____

```
third = 1/3 # needed for statements below
twoThirds = 2/3

'{0:g}'.format(third)
```
_____

```
'{0:g}'.format(twoThirds)
```
_____

```
'{0:.3f}'.format(third)
```
_____

```
'{0:.3f}'.format(twoThirds)
```
_____

```
'{0:f}'.format(3.140)
```
_____

```
'{0:g}'.format(3.140)
```
_____

Now lets look at the 0 before the colon. You can actually list several values to format within the parentheses, each separated by a comma. The 0 refers to the first value listed, 1 is the second etc. (programmers generally start counting at 0). If there is just one value, the 0 is optional - the code will work just the same without it.
Type the following statements in to your file and record the output.

```
print('{0:g}'.format(3.140,3.528))
```
_____

```
print('{1:g}'.format(3.140,3.528))
```
_____

```
print('one third {0:.2f} and two thirds {1:.3f}'.format (third,twoThirds))
```
_____

The example below shows the constant math.pi being formatted to 3 decimal places.

```
import math       # makes functions and constants from the math module of the
                  # Python standard library available to the program
print ("Pi is approx {0:.3f}.".format(math.pi))
```

Type the code below to your file and run it. What is the output?

```
formatted = '{0:g}'.format(3.140)

print(formatted)
```
_____

```
print(formatted * 3)
```
_____

What data type is the value stored in the variable `formatted`? _____

Note that because the formatting operations return strings, there is no need to also turn your number value into a string for concatenation e.g.

```
use      print("Temperature " + str(celsius))
or       print("Temperature " + "{0:.3g}".format(celsius))
but NOT  print("Temperature " + str("{0:.3g}".format(celsius)))
```

The example below shows one way to format a value representing money, with the $ inserted by the programmer, and the value rounded to 2 decimal places.

```
#simple dollars
dollars = 3.4
print('${0:.2f}'.format(dollars))
```

Python also has the facility to format a percentage.

```
#percentage
score = 8
max = 14
print('{:.2%}'.format(score / max))
print('{:.1%}'.format(score / max))
```

Get some practice with formatting by adapting your answers to problems 1, 2 and 4 of this chapter so the output for each is appropriately formatted.

4.5 Practical

Problem 5:

Adapt the circumference program you wrote in Chapter 3 Problem 9 so it formats all the outputs to 2 decimal places.

```
First circle pair:1cm 101cm
Smaller circle's radius: 0.16
Larger circle's radius: 16.07
First circle pair perimeters are 15.92 cm apart.
```

etc.

Problem 6:

Adapt Chapter 2 Problem 11(I have $67.00 to buy lunch for 12 people. How much can each person spend?) so the output is nicely formatted as currency.

```
$67.00 for 12 people
Each person can spend: $5.58
```

Problem 7:

Adapt the see-saw balancing program you wrote in Chapter 3 Problem 7 so it takes the weight of the second person as input from the user. Give the program flexibility by making sure the program will work if a weight like 92.5 is entered. Display your answer to 1 decimal place e.g. Distance from fulcrum for 92.5 kg person is 99.5cm

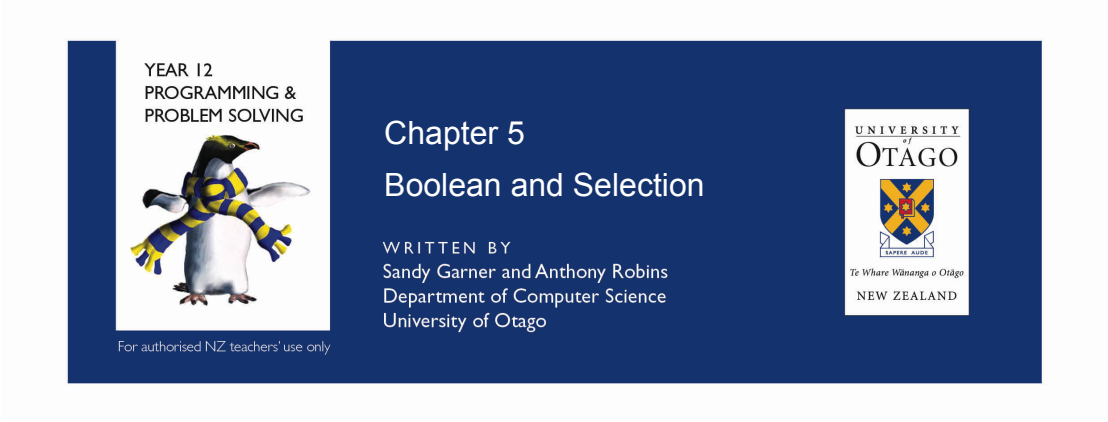
Problem 8:

Write a conversion program of your own choice. It might be from bushels to gallons, from cords to cubic metres, from gallons to litres, from pecks to quarts, from gills to mls, from hogsheads to litres, from euros to yen, from Harry Potter knuts to galleons or sickles . . .

The input should come from the user and the output should be appropriately formatted.

Problem 9:

Write a formatMoney function which takes the number to be formatted as a parameter and returns a string formatted to 2 decimal places, preceded by a dollar sign. e.g. an input of 11.5 would return $11.50



5.0 Introduction

If I am in deep water, swim.

This makes perfect sense to a person. We immediately understand what is meant by deep water. A computer program however would need a clearer specification. What is deep? More than 1 metre? More than 1.5 metres? Over my head? Greater than the distance of my nose from the ground? Greater than an average height? (If the latter, are we assuming an adult height? If so, a male or female adult?)

The solutions to many programming problems require an action to occur only if a particular condition exists. A condition is an exact specification resulting in a value which is unambiguously either true or false. An example of a specification for a condition (given numerical variables `depth` and `myNoseHeight`) might be:

depth >= 1.75

or depth > myNoseHeight

The true or false value which results from the expression of a condition is called a Boolean value (after the mathematician George Boole).

This chapter will cover the construction of conditions, and their use in selecting choices for a program's behaviour.

5.1 Conditions

A condition must result in a value which is either true or false. The reserved words `True` and `False` represent these values in Python. A condition often involves an expression which compares one value with another. Comparisons between numerical values are made using the same comparison operators that are used in mathematics:

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

and the equality / inequality operators:

==	equal to (at last, a proper equals operator)
!=	not equal to

Boundary cases

The difference between >= and > is very important. If you mean "**every child aged 5 and over**" you can express this as **age > 4** or **age >= 5**. Any lack of clarity needs to be removed at the design phase. Does "**every child over 5**" mean "**aged 5 and over**" or "**aged 6 and over**"? Always clarify before coding. The boundary cases are where errors in programming often occur. Boundary cases are the values at and just outside the specified limits. In the case "aged 5 and over" the specified limit is 5 and the age values of `4` and `5` are the boundary cases.

Try This Exercise 1

Write expressions which represent the condition as described, using the variables named where appropriate. Some of the expressions will evaluate to True and some will evaluate to False depending on the values stored in the variables. The first one is done for you.

```
minutes = 50
price = 10.5
temperature = 37.9
```
score = 97

Description	Condition expression	Evaluates to
`minutes` is under 60	`minutes < 60`	True
`minutes` is equal to 50	_____	_____
`price` is over 5	_____	_____
`price` is not over 5	_____	_____
`score` is less than or equal to 50	_____	_____
`score` is not equal to 100	_____	_____
`temperature` is equal to 37.9 degrees	_____	_____

The logical and operator

Let's think about this last expression. Remember that the representation of real numbers is imprecise in programming languages. Here we are testing for a very specific and precise value and we are unlikely to be satisfied with the answer. Float values should not usually be compared using `==` or `!=`.

A better way to evaluate whether a temperature equals 37.9 degrees is to test whether the temperature is within close range e.g. 37.8 to 40.0 degrees. To do this, the code would require two comparisons. Two comparisons can be combined using the **`and`** operator. The **`and`** operator requires that **both** conditions must be true for the expression to evaluate to True.

```
temperature >= 37.8 and temperature < 40.0
```

This expression reads "temperature is greater than or equal to 37.8 and temperature is less than 40". In Python, this expression could also be written as:

```
37.8 <= temperature < 40.0
```

5.2 Selection using if

Conditional expressions are not much use on their own, but they can be put to use in selection and repetition control structures. The first example that we will consider is the **`if`** statement, which has the effect of selecting whether a statement or block of statements will be executed. If the condition evaluates to True, the statement/ block of statements will be performed. If the condition evaluates to False, the statement/ block of statements will be bypassed, and the flow of control will move on to the next statement.

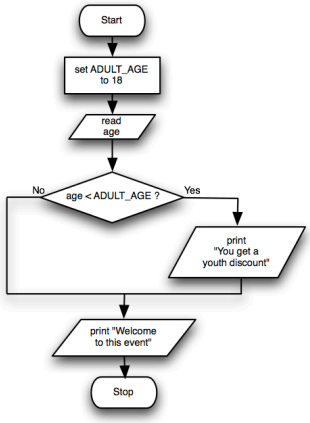
In the example below, `ADULT_AGE` is a constant, and the value stored in `age` comes from user input. The **`if`** condition tests whether `age` is less than `ADULT_AGE`, and if so, displays a message about a youth discount. If `age` is greater than or equal to `ADULT_AGE` this message will not be displayed. In either case the program moves on to the next statement and displays the welcome message.

```
#event welcome
ADULT_AGE = 18
age = int(input("What is your age?"))
if age < ADULT_AGE:
    print("You get a youth discount.")
print("Welcome to this event.")
```

The **if** statement is followed by the conditional expression then by a colon. The statement to be executed if the condition is True is indented.

Flowcharts and selection

Until now, any flowchart designs have been a single line of arrows from Start down to Stop. This is sequential flow of control. With selection there is a choice of paths to follow. A flowchart can clarify visually exactly what should be happening after each decision. The *event welcome* program fragment previously described in words then code could be represented in a flowchart design like this.



Single start point

Process: ADULT_AGE set to 18

Input: age obtained from user

Decision: age < ADULT_AGE

if Yes, Output ...*discount*...

Output: *Welcome*...

Single stop point

Blocks

Anywhere you can use a single statement, you can alternatively use a block of statements. A block is created by indenting statements to the same level. Such a block will only be performed if the condition evaluates to True.

The extended *event welcome* example below has two statements in the **if** block. If you want to execute more than one statement when a condition is true you can follow the **if** condition with a block of statements.

```
#event welcome extended
ADULT_AGE= 18
age = int(input("What is your age?"))
if age < ADULT_AGE:
    print("You get a youth discount.")
    print("You may not purchase alcohol.")
print("Welcome to this event.")
```

Run this example. What is the output if `age` is greater than `ADULT_AGE`?

What is the output if `age` is less than `ADULT_AGE`?

What are the boundary cases for `ADULT_AGE`?

The statements in the block need to be indented to the same level. When the indenting finishes, the **`if`** block finishes. Test this example again (for different values of age) with the indenting removed from the print statement regarding alcohol.

5.3 Practical

Problem 1:

For the situations described, determine the constant value, and the variable required. Give them names (and values where appropriate) then formulate an **`if`** statement with a conditional expression which will evaluate to `True` if the condition is met. Lastly, identify the boundary cases.

a) Is a car's speed within the city speed limit (50kph).

constant declaration _____

variable name _____

condition _____

boundary cases _____

b) Does the number of passengers (unknown) exceed the number of seats (45)

constant declaration _____

variable name _____

condition _____

boundary cases _____

c) Is there 300ml of liquid in the bowl?

constant declaration _____

variable name _____

condition _____

boundary case _____

Problem 2:

Design a flowchart for a program which gets a person's age from the user. If the person is old enough to apply for a Learner driver's licence (16), display "Can apply for Learner Licence". Two sample outputs are shown below.

```
How old are you? 8                (note no output shown for this value)
How old are you? 18
Can apply for Learner Licence
```

List some integers which you could use to test this program. You should test with at least one high value, at least one low value and the boundary cases. Write the program in code. Test your program on your list of test data.

Problem 3:

Design a flowchart for a program which

- gets a person's temperature and age from the user.
- displays a summary of the information.
- if the person is an infant (a child under two) and the temperature is 38 or over, display "Call a doctor".
- if any person of any age has a temperature over 39.5 display "High Fever".

Test your program on a variety of inputs, including the boundary cases. A sample output might look like this:

```
How old is the patient? (years) 1.5
What is the patient's temperature? (celsius) 38.5
Patient aged 1.5 years, with a temperature of 38.5
degrees celsius
Call a doctor
```

Problem 4:

The following code demonstrates the issues involved with comparing real numbers as represented by float values. Type this code into a file.

```
TARGET_TEMP = 37.0
temperature = 36.5 + .1 + .1 + .1 + .1 + .1
if temperature == TARGET_TEMP:
    print("Temperature equals ", TARGET_TEMP, end=' ')
print(" Temperature stored as float", temperature)
```

What would you **expect** this code to print out when it is run?

Run the code. What does it **actually** print out?

Change the condition in the example above so that instead of testing for equality it tests for being reasonably close to a target of 37.0 (set the precision at 6 significant digits after the decimal point 37.0 ± 0.000001).

What will the lower limit of the acceptable range be? What will the upper limit of the acceptable range be? Calculate these values and store them in variables rather than hard-coding them into the expression.

```
Temperature equals 37.0
Lower limit: 36.9999999
Upper limit: 37.0000001
Temperature stored as float: 37.00000000000001
```

5.4 if else

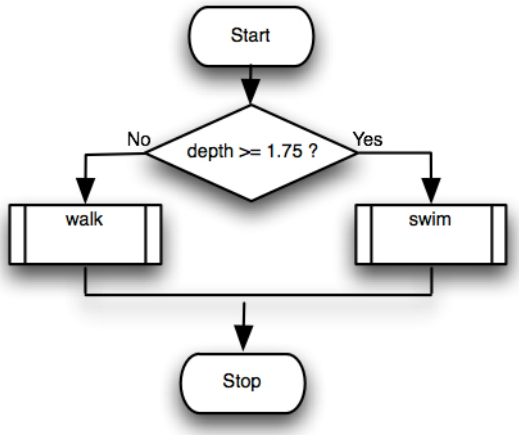
Making choices using just an **if** statement is fine for problems that only need something to be done if a condition is true. Many real-life choices require something else to be done if the condition is not true. This is quite easily achieved in code using the **if..else** structure. Using the deep water example we started with, now we can offer an alternative behaviour:

in English: **if** I am in deep water, swim, **else** walk.

in pseudocode:

```
if  depth >= 1.75
        swim  (this might be a call to a function with a sequence
                of instructions describing a swimming motion)
else
        walk
```

flowchart representation:



This is relatively easy to understand as the meaning of the **if** and **else** matches the use of these words in English. When combined with sensible descriptive variable and function names, the code should be quite clear. If the condition is true, the statement/s in the **if** block are executed, otherwise the statement/s in the **else** block are executed. The processes described in **walk** and **swim** are unknown to this flowchart, but could each be described in a flowchart of their own. As long as we know what the process requires as input and what it gives as output we do not need to know how it works.

When designing a solution to a complex problem, dividing it into manageable portions is sensible. Break the problem down into smaller problems. Each smaller problem can in turn be broken down. This method of solving a problem is often called a **top-down** approach. The top and simplest level is to define the inputs, process them and show the outputs. The next step might be to break them into smaller pieces, and so on.

Try This Exercise 2: Write pseudocode and a flowchart to represent the following situation:

> if age is less than or equal to 10 years, race length is 100m otherwise race length is 200m.
>
> all competitors run the race distance

Here is a coded example of an **if else** statement.

```
#Blood donor eligibility
age = float(input("Age:"))
weight =  float(input("Weight in kgs:"))
DONOR_AGE_LIMIT = 16
DONOR_WEIGHT_LIMIT = 50
if age > DONOR_AGE_LIMIT and weight > DONOR_WEIGHT_LIMIT:
    print("Eligible to donate blood")
else:
    print("Not eligible to donate blood")
```

It is possible for many conditions to be tested in the one expression. The blood donor example could be improved by including the upper age limit (71). The lines below suggest the changes required.

```
DONOR_AGE_LO_LIMIT = 16
DONOR_AGE_HI_LIMIT = 71
if age > DONOR_AGE_LO_LIMIT and age < DONOR_AGE_HI_LIMIT
                                    and weight > DONOR_WEIGHT_LIMIT:
```

Each **and** is evaluated in turn, the result of the first evaluation becoming the left hand side of the next.

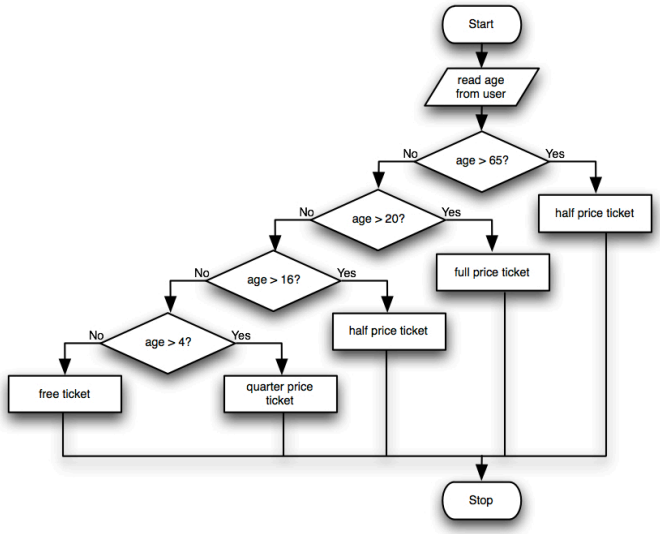
5.5 else if

If your solution needs more than a simple choice between 2 courses of action, a series of **else if** statements might be useful. The example below specifies the price of a ticket depending on age.

Pseudocode example

```
get age from user
if  age > 65
        half price ticket
else if age  > 20
        full price ticket
else if age  > 16
        half price ticket
else if age > 4
        quarter price
ticket
else
        free entry
```

Flowchart example



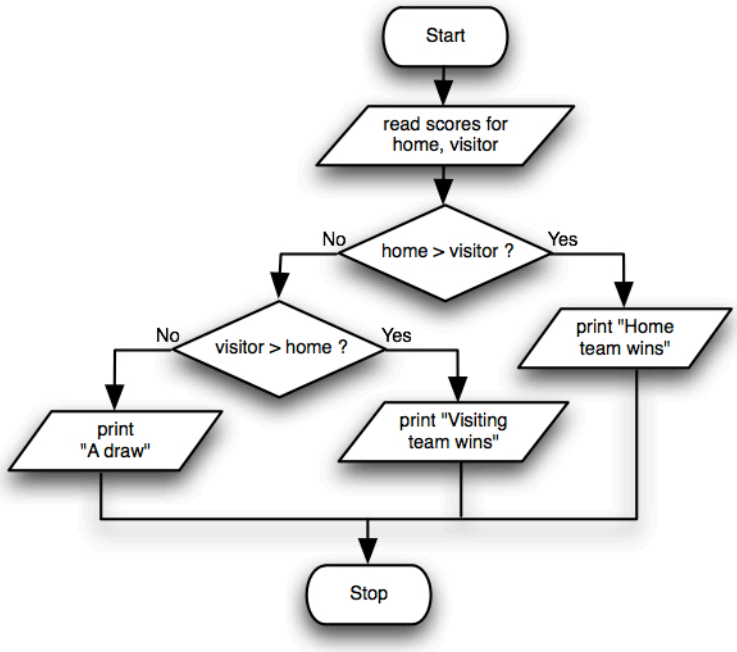
In such a structure only the statement / block following the first true condition encountered will ever be executed. Trace through the flowchart for an age of 25. The age > 20 condition is the first true condition, so "full price ticket" will be executed. It doesn't matter that some of the later conditions are also true as they are not reached.

If no condition is true the statement / block following the final **`else`** is executed. This is called the **default condition**. In this example, an age of 2 would be given free entry via the default condition.

The statements within each **else if** block can be as simple or as complex as required by the solution. There may be just one print or assignment statement, or there could be a sequence of statements including calls to other functions.

In Python, the keyword `elif` represents the **else if** concept.

Here is a flowchart and coded example of an **if else if** structure which displays the origin of the winning team, or a draw, as appropriate.




```
#Display winning team
homePoints = int(input("Score for home team:"))
visitorPoints = int(input("Score for visiting team:"))
if homePoints > visitorPoints:
    print("Home team wins")
elif visitorPoints > homePoints:
    print("Visiting team wins")
else:
    print("A draw")
```

Note there is no need to write another **if** statement to test for the draw - it is the only other possibility.

5.6 Practical

Problem 5:

The speed limit is 50 km/h. Drivers travelling over the speed limit earn demerit points according to the following criteria:

km/h over speed limit	Demerit points
1-10 km/h	10 demerit points
11-20 km/h	20 demerit points
21-30 km/h	35 demerit points
31-35 km/h	40 demerit points
36 km/h and above	50 demerit points

Design and write a program which asks for the speed the car was travelling, and calculates the demerit points earned. Make sure your program behaves correctly for a law-abiding motorist too. A sample output might be:

```
What speed was the car travelling? km/h
80
Speed was 80 km/h
Speed limit is 50 km/h
You were driving 30 km/h over the speed limit
Demerit points: 35
```

How easy would it be to adapt the program you have written to cope with a speed limit other than 50 km/h? If it is easy, well done - your program is flexible.

Problem 6:

Ask for the user to type in an exam mark. Display the mark and a message of **Excellent** for a mark of 85 or above, **Merit** for a mark 65 to 84, **Achieved** for a mark 50 to 64 inclusive, **Not Achieved** for a mark 1 to 49 and **Absent** for a mark of 0.

Use a modular design, with the exam mark as an input parameter. A sample output might be:

```
What was your exam mark?
 34
Mark of 34 : Not Achieved
```

5.7 nested if statements, nested if .. else statements

A sequence of **if else if** statements will evaluate the conditions in order until the first true condition is found, then execute the associated statement / block.

It is also possible to place an **if** statement (or an **if else** statement) inside another **if** or **else** statement. This is called "nesting" the **if** statements (like nesting dolls, where one is completely contained inside another).

```
if feesOwed <= 0
     if score >= 50  this if else is nested inside the first (outer) if
          print "pass"
     else
          print "fail"
else
     print "debtor"
     if age < 18     this if else is nested inside the outer else
          print "parent/caregiver invoiced"
     else
          print "student invoiced"
```

In the pseudocode example above, if a student owes nothing, they will see their exam result (***pass*** or ***fail***).
Otherwise (a student hasn't paid their fees) they will see debtor then depending on their age a message about who was invoiced .
The **if** condition relating to `score` is not evaluated when fees are owed. The flow of control does not reach that point.

The same result could be achieved without nesting by the pseudocode example below. Some program efficiency is lost (two evaluations of `feesOwed` instead of just one) but the code is clearer to follow.

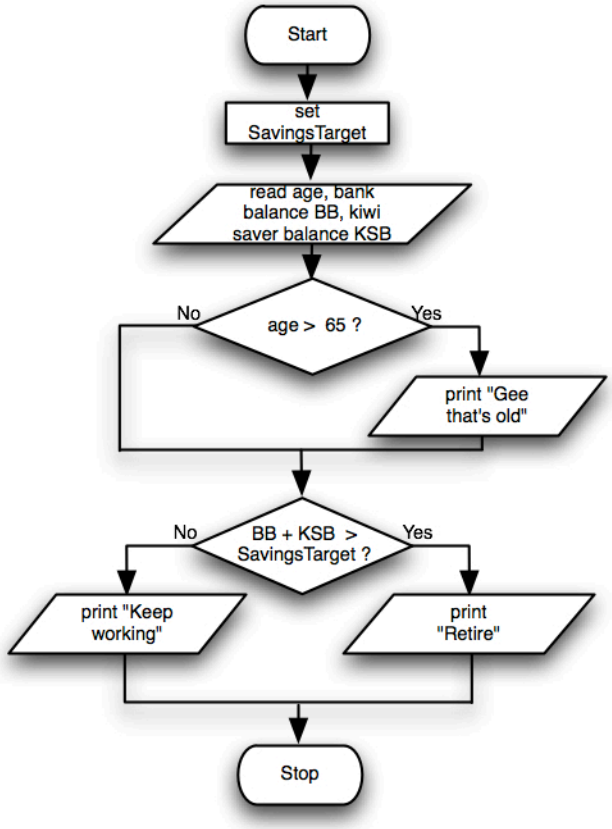
```
if feesOwed <= 0 and score >= 50
     print pass
else if feesOwed <= 0 and score < 50
     print fail
else
     print debtor
     . . .
```

It is important to plan the **if else** structure of code carefully as the meaning of the code will alter with different layouts.

The Retirement programs below are coded examples which demonstrate the difference in meaning possible with different nesting of the **if else** statements. Each program will use the same constant and variable declarations but a different **if** block. There is a flowchart and description for each variation.

Retirement code listing - retirement1 version

```
#Retirement message
age = int(input("Age:"))
bankBalance = float(input("Bank balance: $"))
kiwiSaverBalance =  float(input("Kiwi Saver balance:  $"))
RETIREMENT_AGE = 65
SAVINGS_TARGET = 300000
#retirement1
if age > RETIREMENT_AGE:
    print("Gee that's old")
if bankBalance + kiwiSaverBalance > SAVINGS_TARGET:
    print("Retire")
else:
    print("Keep working")
#end retirement1
```



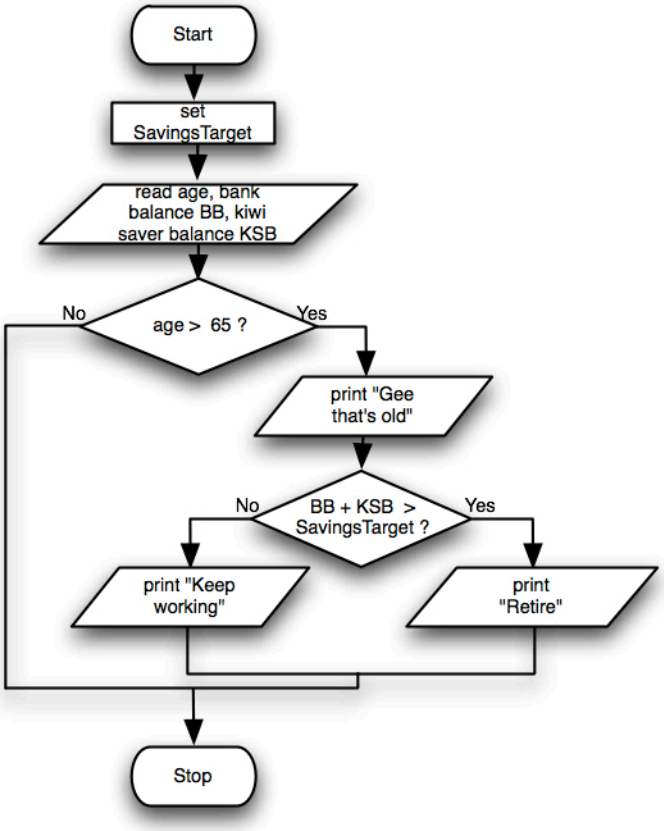
The retirement1 code can have people retire at any age if they have enough money.
There are two separate **if** statements. The second has an **else**.

If they are over 65 they will see ***Gee that's old.***

If they have enough money, they will see ***Retire*** otherwise they will see ***Keep working***.

Replace the retirement1 block of code with the retirement2 block listed below.

```
#retirement2
if age >= RETIREMENT_AGE:
    print("Gee that's old")
    if bankBalance + kiwiSaverBalance > SAVINGS_TARGET:
        print("Retire")
    else:
        print("Keep working")
#end retirement2
```



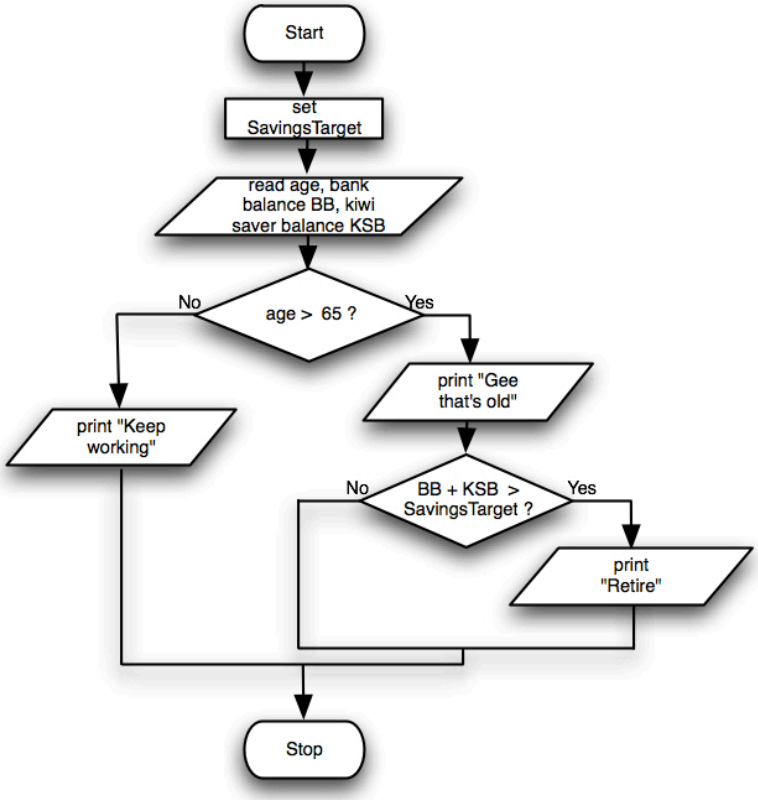
People can only retire at 66 with the retirement2 code. The second **if** statement is nested inside the first **if** statement. The **else** belongs to the second **if** statement.

People 65 and under receive no message at all. People over 65 see *Gee that's old*. If these people have enough money they see *Retire*, otherwise *Keep working*.

Design issue: Only people over 65 have their savings evaluated, so should this information be collected at all for other people?

Now replace the retirement2 block with the retirement3 block listed below.

```
#retirement3
if age >= RETIREMENT_AGE:
    print("Gee that's old")
    if bankBalance + kiwiSaverBalance > SAVINGS_TARGET:
        print("Retire")
else:
    print("Keep working")
#end retirement3
```



The retirement3 code has the second **if** statement nested inside the first **if** statement. The **else** belongs to the first **if** statement.

People can only retire if they are over 65. Otherwise, all younger people are told to ***Keep working***

All the older people see ***Gee that's old***.

Those older people with sufficient savings see ***Retire***, those without have no instruction.

It is important to plan the **if else** structure of code carefully and test both your plan and your code systematically with a variety of data which is selected to cover a range of possible inputs. The test data should test every possible path of the flowchart, and include all the boundary cases.

Problem 7:

The theme park Rainbow's End has a list of person specifications for each ride. Some of these are listed below. Using the information below, design and write a program which will ask for a person's height and age and present them with a list of rides they are eligible to ride on.

FOR CHILDREN 9 YEARS AND UNDER
Dragon Flight Coaster
Freddo's Aeroplane Ride
Car Wash Convoy
Bouncy Castle – children must be 100cm or under
Jumpin' Star – children must be at least 90cm
FOR THOSE 80CM+
Goldrush – a responsible person must accompany those under 100cm
Log Flume – a responsible person must accompany those under 100cm
Pirate Ship – a responsible person must accompany those under 100cm
Dodgems – as passenger only, a responsible person must accompany those under 120cm
Family Karts – as passenger only, a responsible person must accompany those under 150cm

FOR THOSE 120CM+
FearFall – maximum height 195cm
Corkscrew Coaster
Bumper Boats
Dodgems
Invader
FOR THOSE 150CM+
Scorpion Karts
Family Karts
ALSO SUITABLE FOR ALL AGES AND HEIGHTS
Mini Golf
Cinema 180

A sample output might be:

```
What is your height in cm?
125
What is your age?
12
Person 12 years and 125.0 cm high may choose:
Mini Golf
Cinema 180
Goldrush
Log Flume
Pirate Ship
Family Karts as passenger
Corkscrew Coaster
Bumper Boats
Dodgems
Invader
FearFall
```

As you are planning your solution, can you see something wrong with these specifications? What sort of person might be offended by them if they were applied exactly? How could the problem be fixed?

5.8 Boolean variables

A condition evaluates to either true or false and in Python True and False are reserved words representing these values. The result of any expression which can be evaluated to `True` or `False` can be stored as a Boolean variable, which has the data type `bool`.

```
hasPaid = feesOwed <= 0
print("hasPaid is ", hasPaid)
```

If the expression `feesOwed <= 0` evaluates to True, the variable `hasPaid` will hold the value True. If the expression evaluates to False, `hasPaid` will hold the value False.

Remember that you can get Python to tell you what data type the variable `hasPaid` is using the instruction `print(type(hasPaid))`.

The single statement `hasPaid = feesOwed <= 0` has the same effect as the four lines of code below.

```
if feesOwed <= 0:
      hasPaid = True
else:
      hasPaid = False
```

Using Boolean variables has two potential advantages. For one, it can save having to re-evaluate a condition repeatedly (more of an issue when we start using loops). Another is that a sensibly named Boolean variable can improve the readability of your code.

`if hasPaid and hasPassed:` is clear and easy to understand compared with

`if feesOwed <= 0 and score > 50:` which requires some decoding as you read it.

Note that it is **not** necessary to check that a Boolean value is true in this way:

```
      if hasPaid == True and hasPassed == True:
```

The `== True` is redundant, as the variable is simply `True` or `False` without it.

Problem 8:

To apply for your restricted licence, you must:

- **be at least 16½ years old**
- **have held your learner licence for at least six months**

Write a program which asks whether the two relevant pieces of information from the user are true. Display a suitable message letting the applicant know whether he/she is able to apply for a restricted licence. Rather than cope with all the things a user might type intentionally (Y, y, Yes, yes, YES) or unintentionally (yess, Ye, Yss) just ask for a **1** to represent **yes** and a **0** to represent **no**. A sample output might be:

```
Have you held your Learner licence for six months?
1 yes
0 no
1

Are you 16 and a half years old?
1 yes
0 no
1
You can apply for Restricted licence.
```

5.9 Boolean return modules

A function can return a Boolean value. A call to a return function can be used as all or part of an expression to be evaluated.

```
def isTall(height):
      TALL = 1.5
      if height > TALL:
            return True
      else:
            return False
#main routine
heightInCm = int(input("What is your height in cm?"))
if isTall(heightInCm):
      #some code here which applies to tall people
```

Using a Boolean return function might be appropriate if you don't want to store the value. Let's imagine that to determine whether a student has paid fees requires a series of requests/access to other modules, and perhaps a real-time access to the cashier's database. Let's assume that a function called `hasPaid` takes care of all this and just returns to us the True or False result. We can use the function call as part of the condition e.g.

```
if hasPaid() and hasPassed:
```

Is `hasPassed` a function or a variable? How can you tell?

5.10 logical or, logical not, truth tables

The **and** operator only operates on Boolean values, and is called a logical operator. There are two other logical operators.

logical or

Two comparisons can be combined using the **or** operator. This means if **either** of the conditions is true the result will be true. Let's assume a Boolean variable called `level3NCEA` has stored whether a students has sufficient NCEA credits to attend university, and `age` is an integer variable. The statement below will set `mayAttendUni` to True if either of the conditions is True.

mayAttendUni = age >= 20 `or` level3NCEA

logical not

Sometimes it is clearer to think in terms of what isn't true. If you have a Boolean variable called `chosen`, you could select the case of *not* chosen using

```
if not chosen:
```

Truth table demonstrating logical and, or and not.

p	q	p and q	p or q	not p
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Try This Exercise 3: Given the variable declarations below, fill in the truth table making sure you cover all possible combinations of `adult` and `licence`.

```
adult = age >= 18
licence = hasDriversLicence()#assume function returns a boolean value
```

adult	licence	adult and licence	adult or licence	not licence

Operator Precedence

The pseudocode condition "not adult and licence" is ambiguous – not(adult and licence) will give a different result from not(adult) and licence. We need precedence rules to define the outcome predictably.

Evaluated first

`* / %` (more than 1 are performed in left to right order)

`+ -` (more than 1 are performed in left to right order)

The relational operators `< > >= <=` are performed in left to right order.

The equality operators `== !=` are performed in left to right order.

The logical operator **not**

The logical operator **`and`** (more than 1 are performed in left to right order)

The logical operator **`or`** (more than 1 are performed in left to right order)

The assignment operator = happens last

Evaluated last

The use of parentheses to clarify precedence is advised. As in mathematics, parentheses can also be used to override precedence.

Try This Exercise 4

More complete options for University Admission are that a student must have satisfied one of

a) age at least 20
or
b) level3NCEA and age at least 16

and all students must be competent in written and spoken English.

The code below tries to summarize the university admission regulations, but produces the wrong result for the data.

```
age = 25
level3NCEA  = True
english = False
print(age >= 20 or age >= 16 and level3NCEA and english)
```

Why is the code producing the wrong answer?

How could you fix it?

Developing a set of test data

When writing a logical condition, you should develop a set of test data you will use to check that you haven't made an error.

The test data should include:

- values which try out each line of the truth table i.e. each possible combination of true and false.
- values which are clearly inside each condition.
- values which are on and immediately outside of each numerical boundary value.
- unexpected values e.g. of the wrong data type.

There is always more than one way to express a condition. Some are easier to follow than others.

Try This Exercise 5

How many ways can you rewrite this expression while keeping the meaning the same (and assuming score is an integer)?

score > 3 `and` score < 5

Which of these do you think is the clearest to read?

There is a fine line between clever, concise code and clear, readable code. It is better for your code to be readable rather than short, but your code should not be longer than is necessary for clarity. Generally speaking if you find you are evaluating the same condition more than once you should check that the second and subsequent evaluations are necessary.

5.11 Practical

Problem 9:

```
LIMIT = 300
depth = int(input("Current depth?"))
#marker
if depth >= LIMIT:
      full = True
else:
      full = False
```

Rewrite the code following the marker comment as a single line assignment statement.

Problem 10:

Write an **if** statement which will test whether a number typed in by the user is an illegal exam mark (outside the range 0 to 100, inclusive). Display the mark, and a message showing whether it is legal or illegal. Sample output might be:

```
Type in a number  105
105 is not a legal exam score
```

Problem 11:

Integrate the code from Problem 10 into your Problem 6 solution, so that an illegal score is not processed.

Develop a set of test data, and test your program thoroughly.

Problem 12:

The recommended 4-6 hourly dose of paracetamol for children under 12 years is 10 milligrams of paracetamol for every kilogram of body weight. For adults and children 12 and over, it is two 500milligram tablets. Write a program which collects the necessary information about the user, and suggests a dose of paracetamol. An example of output might be:

```
What is the patient's age? 9
What is the child's weight? (kg) 35
Recommend  350  mg paracetamol
```

Problem 13:

On Monday to Friday the alarm must ring at 7, on Saturday at 8 and on Sunday at 9. If the days of the week are represented by numbers (Monday = 1 etc.) write a program which asks what day of the week it is and displays the alarm time.

```
What day is it?
1 Monday ... 7 Sunday  1
Alarm rings at 7
```

Research topic: Who was George Boole and what was his legacy?

5.12 Python's doctest demonstrated

Now that we are using selection in our programs, testing them thoroughly is more important than ever. The doctest module from the Python libraries can be really useful for checking that a function is working reliably. To use doctest we are going to need to include a **docstring** at the beginning of the function's body. A docstring is just a triple quoted string literal. Inside it you can list all the expected cases, boundary cases and "in error" cases that you want to be tested as you develop your solution and every time you alter your code. Once you have put everything in place (see below for details) these tests will be made automatically every time your program is run.

To use the doctest function, follow these steps (a code example follows on the next page):

Run the doctest class's testmod function:

1. Put the statement `import doctest` at the bottom of your `main` routine or at the top of your program.

2. In the `main` routine, the statement `doctest.testmod()` will run the doctest. If all the tests pass we will simply get the output we normally would without them. If one or more of them fail we will also get a report showing which ones are not currently passing. Alternatively, the statement `doctest.testmod(verbose = True)` will run and document the doctest anyway, even if all the tests all pass – use this option while the program is in the development and testing phase.

List the test conditions inside the function containg the code to be tested:

3. The line immediately below the `def` header should contain 3 triple quotes `"""`
This is the beginning of the docstring.

4. The next line contains 3 greater than symbols `>>>` (the prompt) then a space and finally a call to the function, including the input parameter/parameters to be tested.
There **must** be a space between the prompt >>> and the function call.

5. The line immediately below the prompt >>> line is for the expected output/result of the function with that/those particular input parameter(s). The output might be a print statement or statements, or it might be a return statement, or both.
If function is **printing out** a string, do not use any quotes for the expected result.
If the function is returning a string, use **single** quotes.
If the function is printing out a result that takes more than 1 line, format the expected output in the same way.
Steps 4 and 5 can be repeated as often as desired, one pair of statements for each test case (expected, boundary etc.)

6. Close the docstring with another triple quote `"""`

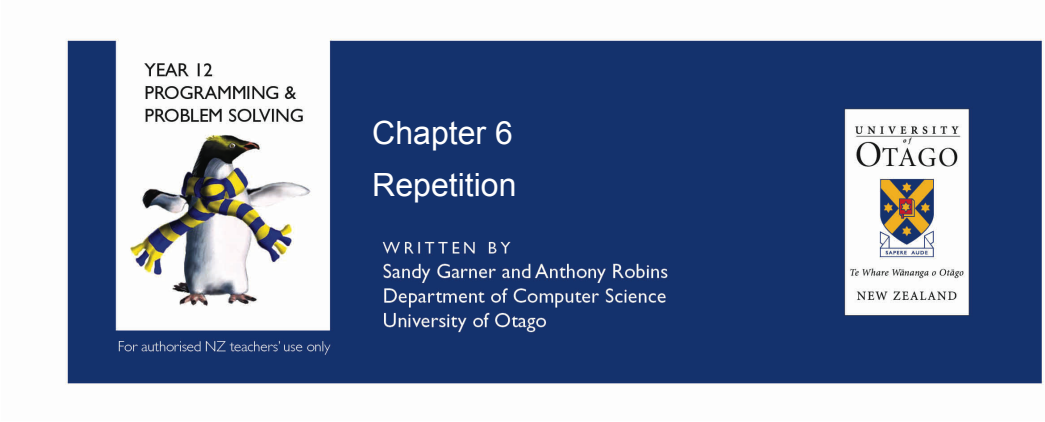
7. Below that, write the body of your function in the same way that you would have without the doctest. (This may in reality be the first step.)

doctest example code

```
#function to multiply and return 2 numbers
def multiply(num1, num2):                    #function header
    """                                      #step 3
    >>> multiply(1,2)                        #step 4
    2                                        #step 5
    >>> multiply(0,2)                        #step 4
    0                                        #step 5
    """                                      #step 6
    return num1 * num2                       #step 7
#main routine
print(multiply(4,6))
#these next lines can go at the bottom of the program, out of the way
import doctest                               #step 1
doctest.testmod(verbose = True)              #step 2
```

A more extensive doctest example

```
#function to add and return 2 numbers
def add(num1, num2):
    """
    >>> add(1,2)
    3
    >>> add(-1,1)
    0
    """
    return num1 + num2

#function to return a string describing stage of life - note single quote around string
def kidult(age):
    """
    >>> kidult(9)
    'child'
    >>> kidult(13)
    'teenager'
    >>> kidult(21)
    'adult'
    >>> kidult(-1)
    'invalid'
    """
    if 0 <= age < 13:
        return "child"
    elif 13 <= age <= 20:
            return "teenager"
    elif 20 < age < 125:
        return "adult"
    else:
        return "invalid"

#function to display a string - note NO quote around string
def team(teamMembers):
    """
    >>> team(0)
    Insufficient
    >>> team(7)
    Insufficient
    >>> team(8)
    Team
    >>> team(10)
    Team
    >>> team(12)
    Too many
    Start another team
    """
    if teamMembers <= 7:
        print("Insufficient")
    elif 7 < teamMembers <=10:
        print("Team")
    else:
        print("Too many")
        print("Start another team")

#main routine
print(team(4))
import doctest
doctest.testmod(verbose=True)
# all the program's doctests are run, even if the function they are in is not called in the program
```



6.0 Introduction

Computation usually involves the repetition of some process. A computer might be processing a stream of numbers from an input device such as a tide gauge, or processing thousands of stored records from a database, or responding to the press of a button in a game. A computer can repeat a task accurately all day long. It just needs to know when to start, what to do and when to stop. To program this repetition, we use a **loop**. In computer science the term **iteration** is often used to mean the repetition of a process. Each time through the loop is called an iteration.

Chapter 5 covered the topic of formulating conditions, and used them in selection structures (if, if..else etc.). In this chapter we use exactly the same kinds of conditions to start and stop loops.

There are various loops which are a part of the Python language. These include the **while loop** and the **for loop** covered in this chapter.

6.1 while loops

The construction of a while loop is really very simple. It specifies a condition and a body (statement/block of statements):

while *some condition is true*:
    perform a statement/block of statements

To prevent a loop from repeating forever, one of the statements must update a value in such a way that the condition will at some point become false. It is good style to make this update the last statement in the loop. Here's a pseudocode example of a while loop (assuming values for `fortune`, `ideal` and `income`):

```
while fortune < ideal:
    work another week
    keep a count of how many weeks
    pay the bills
    add leftover income to fortune
```

To begin with, if `fortune` is less than `ideal`, the loop body will be executed for the first time. The last statement in the body is the update. It increases the value of `fortune`. Then we go back to the top of the loop and check the condition again. If it is still true we execute the body for the second time.

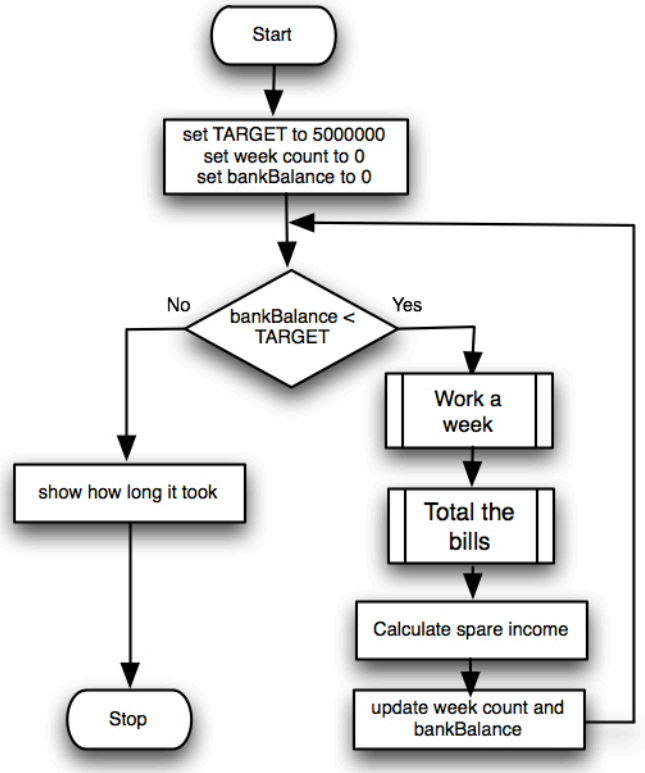
We keep repeating this process until (because `fortune` keeps increasing) at some point the condition is false. Then we're finished with the loop, and move on to the next statement in the program.

If `income` is `0` or negative then `fortune` will not increase and the loop will never end.

If `fortune` is as large as `ideal` to begin with, the loop will never start.

Let's try to imagine the program this idea might be part of and specify it first as a flowchart then as pseudocode. We would need a constant to represent `ideal` - we will call it `TARGET` and set it at 5 million. A variable named `bankBalance` can represent the fortune.

Flowchart



Pseudocode

```
set TARGET to 5000000
set bankBalance to 0
set numWeeks to 0
while bankBalance is less than TARGET
      work_a_week   this will be a module which describes weekly
                    earnings, tax etc. and returns the income available
      total_the_bills  this module will return the total amount of
                       money to be paid out on bills this week
      add 1 to numWeeks
      spareIncome = work_a_week - total_the_bills
      add spareIncome to bankBalance  update the condition
show the number of weeks it took
```

Converting the plan into code.

1. Declare and initialize all the variables and constants which are required before the loop.

```
TARGET = 5000000
bankBalance = 0.0
numWeeks = 0
```

2. Write empty functions called `workWeek` and `totalBills`. For now, just make each function return some float value. We can worry about what goes in these functions later (this is top down programming in action). It is important though to describe the intended purpose of these empty modules in the header comment.

```
#returns weekly income after tax
def workWeek():
    return 2000
#returns total owed this week on bills
def totalBills():
    return 600
```

3. In the main routine, follow the setup variables with a while loop.

```
while bankBalance < TARGET:
```

4. Inside the loop, the important data as far as `bankBalance` is concerned is the spare income. It can be calculated by calling the 2 return functions like this:

```
spareIncome = workWeek() - totalBills()
```

Then increment the week counter and update the bank balance:

```
numWeeks = numWeeks + 1
bankBalance = bankBalance + spareIncome
```

The program can run now, but it will not display anything. What you really want to know is how many weeks it will take you to reach the target. Where should you put the print statement? What would happen if you put it inside the loop?

Probably in this case you will just want a single print out after the loop is formatted.

5. After the **while** loop and before the end of the main routine write the statement:

```
print("Weeks to target", numWeeks)
```

If you run the code now you should see some output:

```
Weeks to target 3572.
```

6. Some adjustment to the plan seems necessary. If the number of weeks is over 52, the display would be more sensible in years. Write an if statement which displays the output in weeks **or** years:

```
if numWeeks > 52:
    print("Years to target", int(numWeeks/52))
else:
    print("Weeks to target", numWeeks)
```

   (If the number of years is small, it would be appropriate to have both years and left-over weeks showing. Feel free to improve the code.)

   What would happen if your **if** statement was inside the **while** loop? Try it if you like. Use Control C if you need to force a loop to finish.

   The **loop control** variable is the variable which is updated – in this case **bankBalance.**

   The **threshold value** is the term we will use for the value which the loop control variable is compared to – in this case **TARGET**

7. What if you wanted to know what your bank balance will be after 6 months? This time you know exactly how many times you want the loop to repeat. Change the while loop condition to `numWeeks < 26` and display the balance with the print line:

```
print("Bank balance after", numWeeks, "weeks is $", bankBalance)
```

   What is the loop control variable now?

   This loop is now a **counter controlled** loop, since `numWeeks` is simply a count of how many times the loop will iterate.

   Why is it better to show the value of `numWeeks` rather than `26` in the print statement?

8. Change the code so it shows how much would be saved in 2 years. Hopefully you only needed to change one number.

Some things to remember when writing loops

If the loop control variable is not updated within the loop in such a way that the condition becomes false at some point, the loop will never finish. Your program will "hang". This is called an infinite loop.

If the loop control variable is being reset to its starting value within the loop, as well as being updated, the loop will only finish if one update is sufficient to reach the threshold value. The loop control variable should be initialized before the loop.

All loops have a body which is a statement or block of statements. As for any other block of statements, any variable declared inside a loop will not be available outside the loop.

Input checking with a loop

A loop can be used to check that the user has entered one of a range of appropriate values.

Imagine a user has been presented with a numbered menu:

1 Play World of Warcraft

2 Play Sims

3 Play Pacman

4 Play Chess

5 Quit

A **while** loop can stop the program proceeding until the user has entered one of the numbers in the desired range.

example 1

```
choice = 0  # needs to be initialised outside of the while loop
while choice < 1 or choice > 5:
    choice = int(input("Choose a number between 1 and 5 "))
    if choice < 1 or choice > 5:
        print("Not a valid choice - try again")
print("Choice is",choice)
```

The **if** statement is optional technically, but is essential information for the user. This code will work as long as the user chooses a number, not any other key.

It is important if the program is to continue on successfully that it has the data it needs. The code above leaves `choice` at 0 if there was an entry error. The user needs another chance. In section 4.3 you learned how to use `try` and `except` to cope with bad data input. A `while` loop may be able to achieve the same purpose.

Below is a variation of example1 above using the Boolean flag `goodChoice` to determine when to leave the loop. A Boolean flag is like a signal flag - either up (on, true) or down (off, false). The flag is toggled by some process within the loop. The loop iterates until the flag becomes true. This code can be very easy to read if the names are chosen carefully.

example 2

```
choice = 0
goodChoice = False
while not goodChoice:  # while goodChoice is false
    choice = int(input("Choose a number between 1 and 5 "))
    if choice < 1 or choice > 5:
        print("Not a valid choice – try again")
    else:
        goodChoice = True
print("Choice is",choice)
```

This code will work no matter what key is pressed - it will keep asking for a number until it gets one within the required range. The message to the user is always the same: "Not a valid choice - try again".

But, the example 2 code would have problems if `0` (the initial value of `choice`) was a valid choice in terms of the programming solution. Bad input will produce an accidental 0. This needs to be different from an intentional 0. The example 4 provides a solution to this problem.

Including try except

Remember from chapter 4.3 the statements which will cope with unintended input. The code which might cause an error is put in a `try` block. If there is an error, the code in the `except` block will be executed and the program will not crash. Here is a version of the code from section 4.3:

example 3

```
#reads an int from the keyboard
choice = 0
try:
    choice = int(input("Choose a number "))
except ValueError:
    print("Not a number")
print("Choice is",choice)
```

The example 4 code below combines the **while** loop from example 2 (**goodChoice**) with the **try except** code from example 3 above. It allows 0 to be a valid input value.

example 4

```
#reads an int from the keyboard
choice = 0
goodChoice = False
while not goodChoice:
   try:
      choice = int(input("Choose a number between 0 and 5 "))
      if choice < 0 or choice > 5:
         print("Not a valid choice – try again")
      else:
         goodChoice = True
   except ValueError:
         print("Not a number – try again")
print("Choice is",choice)
```

This code keeps asking for a number within the required range until it gets one.

Challenge

Design and write a function which accepts a high and low value as parameters and limits an integer returned to be within these values.

6.2 Practical

Problem 1:

You have a secret 4 digit pin (numerical) which you are asked to type in to your program.

A hacker wants to try every number combination until your secret code is discovered.

Python will interpret 0054 as 54. Design and write a program to discover and display the secret pin.

See if you can display the secret pin with any leading zeroes necessary.

```
Enter your secret 4 digit pin 0456
Your secret is 0456
```

Problem 2:

Write a modular program which tests to see whether any integer entered is a prime number.

```
Enter your number 133
133 is not prime
```

Extension:
Ask the user to enter a prime number, keep asking for another number until they do.

Problem 3:

You invest $1000 (the principal). Each year the interest is added to the principal.

Design and write a program which gets the interest rate and number of years from the user, and calculates and displays the principal expected after that time at that interest rate.

```
Enter interest rate 5.75
Enter investment period (years) 10
After 10 years your balance is $1749.06
```

Problem 4:

For each round of a boxing match the winner receives a score of 10 and the other boxer 9, or 0 if he/she is knocked out. The match lasts for 10 rounds or until a player is knocked-out. The total score decides the winner.

Design and write a program which collects the score for each player for each round of a boxing match then displays the total points for each player, the number of rounds and the name of the winner.

```
Round 1
Enter score for Player A 10
Enter score for Player B 9
Round 2
Enter score for Player A 9
Enter score for Player B 10
Round 3
Enter score for Player A 10
Enter score for Player B 0
Player A 29
Player B 19
Rounds fought: 3
Player A is the winner
```

Problem 5:

Design and write a program which uses a loop to ask for integer values from the user, and calculates the average. How many values are to be averaged is the first thing the user is asked for.

```
How many numbers to be averaged? 3
Enter number 1 10
Enter number 2 12
Enter number 3 13
Total is 35
Average is 11.666666666666666
```

6.3 for loop

For loops in Python are very powerful and flexible. Their most common use is based on a simple loop control variable which is a counter. If you know you want a loop to repeat an exact number of times then probably a **for** loop will be the best choice.

The loop control variable, test condition and update are all together in the for loop header but they are a little more obscure than is the case with a while loop.

```
#returns weekly income after tax
def workWeek():
    return 2000
#returns total owed this week on bills
def totalBills():
    return 600
#main routine
WEEKS = 26
bankBalance = 0.0
for week in range(0,WEEKS):   #for loop header
    spareIncome = workWeek() - totalBills()
    bankBalance = bankBalance + spareIncome
    print(week)

print("Bank balance after" , WEEKS, "weeks is $",
                                  "{0:.2f}".format(bankBalance))
```

Python's **range** function lets you specify a sequence of numbers. It starts at the first number and finishes **before** the second number. So (0, WEEKS) in the example above will run from 0 to 25. By default, the range counts up one at a time.

Look at the **for** loop header below. The loop control variable is week. The start condition is week = 0, the stop condition is week < WEEKS and the update is the default of week = week + 1.

```
for week in range(0,WEEKS):
```

What are the boundary cases?

Counting in 2's from 0 to 100

```
for count in range (0,101,2):
```

Counting down from 10 to 1

```
for count in range (10,0,-1):
```

6.4 Practical

Problem 6:

There are 5 races in a sporting event - use a **for** loop to collect and display the winner's names for each event.
(The display statement should be within the loop. Each new name can overwrite the previous value. Storing each name separately for access outside the loop will be easy with a `list`, and will be covered in Chapter 7)

```
Enter winner's name for race 1 Anna
Anna
Enter winner's name for race 2 Bella
Bella
Enter winner's name for race 3 Hanna
Hanna
Enter winner's name for race 4 Lucy
Lucy
Enter winner's name for race 5 Oliver
Oliver
```

Problem 7:

A golf game has 9 or 18 holes. Ask the user which length of game is being played, and write a loop which collects the score for each hole and tallies them.

Problem 8:

If a chessboard were to have 1 grain of rice placed upon the first square, two on the second, four on the third, and so on (doubling the number of grains on each subsequent square), how many grains of rice would be on the last (64th) square?

```
Square 1 Grains 1
Square 2 Grains 2
Square 3 Grains 4
Square 4 Grains 8
Square 5 Grains 16
```
etc.

6.5 nested loops

Loops can be nested inside each other. The algorithm below describes one way of printing all the hours in a day (the numbers 1 to 12 twice). The second loop (to 12) is executed completely for each iteration of the first loop (to 2). This algorithm has not yet defined which kind of loop will be used. The numbers, letters and ii's represent levels of indentation, so a) b) and c) belong inside the first loop, while i) and ii) belong inside the second.

Output will be 1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9 10 11 12

Algorithm

1) set count to 1
2) loop 2 times (loop control is count)
   a) set hour to 1
   b) loop 12 times (loop control is hour)
      i) print hour
      ii) increment hour
   c) increment count

Pseudocode suggesting nested for **loops**

```
set count to 1
for count = 1 to 2, increment by 1
        for hours = 1 to 12, increment by 1
                print hours
```

Pseudocode suggesting nested while **loops**

```
set count to 1
while count is less than 2
        set hours to 1
        while hours < 12
                print hours
                increment hours
        increment count
```

Code using nested for **loops**

```
#for
for count in range (0,2):
    for hour in range (1,13):
        print (hour,end=' ')
```

Code using nested while **loops**

```
#while
count = 1
while count <= 2:
    hour = 1
    while hour <= 12:
        print(hour,end=' ')
        hour = hour + 1
    count = count + 1
```

6.6 Practical

Problem 9:

A robot has a light sensor which reads 60 on the background colour and 30 on a patch of a darker colour. The robot can be programmed to wiggle around the edge of the patch by

> moving the left wheel until the light sensor reads above a chosen threshold then
> moving the right wheel until the light sensor reads below the threshold then
> moving the left wheel until the light sensor reads above a chosen threshold then
> moving the right wheel until the light sensor reads below the threshold then . . .

Assume the robot has `rotateLeft` and `rotateRight` functions which turn the named wheel while stopping the other wheel, and a function called `getLight` which returns the light reading from the sensor.

Design and write an algorithm or pseudocode which

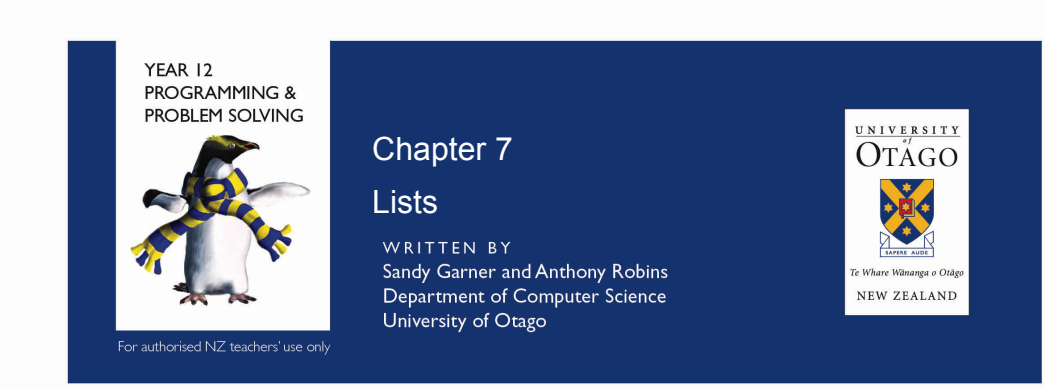
- chooses a suitable light threshold, half way between the two initial readings
- instructs the robot to move around the edge of the dark patch
- uses loops where appropriate
- carries on forever (while(`True`) is the simplest way to implement an infinite loop)

Problem 10:

A clothing firm wants to print out a batch of size labels. They want 100 of each of size 6, 8, 10, 12, 14, 16, 18 and 20. They have a label printing process which is represented by the function below. It takes the size as an input parameter.

Design and write a program which uses nested loops to print 100 of each label size.

```
def printlabel(size):
    print(" _____________________ ")
    print("|Dresswell Clothing Co |")
    print(" ******size " + str(size) + " *****")
    print("|_____________________|")
```



7.0 Introduction

Most of our programs so far have involved just a few pieces of data. In these cases each value can be stored in its own variable. But many real-world tasks involve tens, hundreds, thousands or millions of values. Computers were designed for such repetitive data processing tasks. But it is not practical to process a large amount of data while each individual variable has its own unique name.

That is why programming languages have **indexed** variables, which are contained within indexed data structures such as **lists**. Lists hold lots of values. The items stored in a list share the same variable name, but each item has a different index number, like letter box numbers in a street. Lists go hand in glove with loops and repetition to access the values one at a time as needed.

7.1 Indexed variables

So far we have been working with ordinary variables, which have names such as `score`.

An indexed variable is just a variable name followed by an index number such as `score[1]`. The indexed variable's identifier (name) can reflect the fact that it contains more than one value by being a plural, or a collective noun e.g. `scores` or `scoreList`. An indexed variable can be used in exactly the same way as an ordinary variable. We can assign a value to it:

```
scores[1] = 5
```

We can use it in a calculation:

```
sum = sum + scores[1]
```

We can send its value to a function: `myFunction( scores[1] )`

```
print( scores[1] )
```

And so on. Anything we can do with a variable we can do with an indexed variable.

One big advantage of indexed variables is that we can create lots of them at once, for example:

```
scores = [0] * 100
```

declares 100 indexed variables, `scores[0]` to `scores[99]`, in just one statement (and initialises each one to 0). Here `scores` is called a list, and the individual indexed variables are known as the elements of the list.

Another big advantage is that we can process all the elements in a list easily. If we want to add them up, then a single statement like:

```
sum = sum + scores[i]
```

will work, if it is contained within a loop where `i` is a variable which counts from 0 to 99.

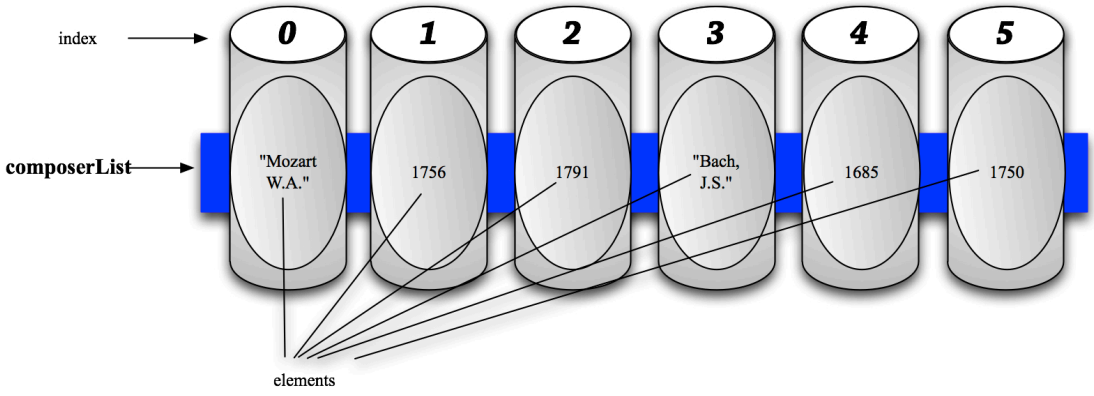
There – you already know the essence of lists! But read on for the details…

7.2 A first look at lists

The main features of lists are:

- A list stores data sequentially
- A list stores data of any mix of data types
- A list has a name
- Each piece of data is called an element
- Each element has a unique index
- The first element is stored at position 0 (index 0)
- Lists can grow and shrink as elements are inserted and deleted.

The diagram below represents a list with the name `composerList`. It stores 6 values, one at each of the index positions 0 to 5. Think of the index as a lid opening into a container. Individual data values can be deposited and retrieved using the index number.



Before you can do anything with a list, it needs to exist and be filled with useful values. The list in the diagram could have been created with the statement:

```
composerList = ["Mozart W.A.", 1756, 1791, "Bach J.S.", 1685, 1750]
```

Individual elements are accessed using the syntax `listName[index]`.

To access the first element of `composerList` (index position 0), use the expression `composerList[0]` e.g.

```
print(composerList[0]) will display Mozart W.A.
```

To access the last element of `the list` use the expression `composerList[-1]` e.g.

```
print(composerList[-1]) # will display 1750
```

To store data in the first position of `composerList` (index position 0), use the syntax `composerList[0] = new_item` e.g.

```
composerList[0] = 100      # will replace the string "Amadeus W.A."
                             with the integer 100
```

Lists have a function **len**, short for length, which returns the number of elements it contains using the syntax `(len)listName`.

```
lengthOfList = len(composerList)
print("Length of list is ", len(composerList))

if len(composerList) < 3:
```

The expression `len(composerList)` will return the value 6, but the indices range from 0 to 5.

7.3 Storing values in a list

List assignment can take many forms. Here are some simple examples of list declarations accompanied by diagrams of each list after the instruction has been performed.

```
numbers = [20, 36, 12, 24, 20, 48, 74, 353, 23, 98]
#declares a list of int values called numbers
```

index	0	1	2	3	4	5	6	7	8	9
value stored	20	36	12	24	20	48	74	353	23	98

```
passed = ["Peter", False, "Amanda", True]
#declares a list of mixed values called passed
```

index	0	1	2	3
value stored	"Peter"	False	"Amanda"	True

The range function is useful for putting a series of numbers in a list.

```
odds = list(range(5,30,2))
# declares a list of int called odds which holds the odd values from 5 to 29
```

index	0	1	2	3	4	5	6	7	8	9	11	12	13
value stored	5	7	9	11	13	15	17	19	21	23	25	27	29

```
empty = []
 #declares a list called empty which holds no values
```

Two lists can be concatenated into a single list.

```
oneMore = passed + ["Ben", True]
#declares a list called oneMore which begins with passed and puts the new list on the end
```

index	0	1	2	3	4	5
value stored	"Peter"	False	"Amanda"	True	"Ben"	True

Elements can be deleted from any position in the list. The length of the list is reduced, and every element after the deleted elements is renumbered.

```
scores = [11, 45, 52, 31, 24]
```

index	0	1	2	3	4
value stored	11	45	52	31	24

```
scores[1:3] = []
#deletes elements starting at position 1, finishing before position 3.
```

index	0	1	2
value stored	11	31	24

A list can be inserted into another list. Every element after the inserted elements is renumbered.

```
count = [1,4,5]
```

index	0	1	2
value stored	1	4	5

```
count[1:1] = [2,3]
#inserts the new list starting at position 1, finishing before position 1.
```

index	0	1	2	3	4
value stored	1	2	3	4	5

An element can be appended to the end of the list.

```
count = [1,2,3,4,5]
```

(diagram as above)

```
count.append(6)
```

index	0	1	2	3	4	5
value stored	1	2	3	4	5	6

Complete the table below, using the lists declared above.

list name	expression to return the length of the list	value returned by `len` function	index of the 1st element	index of the last element
`odds`	`len(odds)`	`13`	`0`	`9`
`empty`			`error`	`error`
`passed`				
`oneMore`				
`numbers`				

Try this

Initialise a list called `evens`, filled with the even numbers greater than 1 and less than 20.

Initialise a list called `dataLog` which is empty to begin with.

Initialise a list called `family`, filled with the names of your family members.

7.4 Using a for loop to fill a list with data

It is possible to fill lists with data as we have seen using a series of values in square brackets, or individually with an assignment statement for each element, but lists are often very large, so it is more usual to fill them with values using a loop.

The example below shows the `numbers` list being repeatedly concatenated with integers typed by the user. The list is initialised before the loop, as an empty list.

```
SIZE = 10
numbers=[]
for n in range(SIZE): #covers values from 0 to 9
    numbers = numbers + [int(input("Enter an integer"))]
print(numbers)
```

The next example shows the `numbers` list being initialized as a list of 0's, which are then replaced in the for loop by integers typed by the user.

```
SIZE = 10
numbers = [0] * SIZE  # all elements are set to 0
for index in range(SIZE):
    numbers[index] = int(input("Enter an integer "))
print(numbers)
```

The next example shows a list being filled by an expression involving the range value.

```
squares = []
for sqRoot in range(1,6):
    squares = squares + [sqRoot * sqRoot]
print (squares)
```

index	0	1	2	3	4
value stored	1	4	9	16	25

7.5 Using a loop to read or process all or part of a list

Lists have a length, and the list index increments by 1, so a **while** loop can represent the index of each element of a list in turn.

```
while index < len(numbers):
      print(numbers[index])
      index = index + 1
```

The example below shows the values in the `numbers` list being displayed using a **for** loop. The syntax is simpler because there is no need for the index variable.

```
#displays each value in the numbers list using a for loop
for number in numbers:
      print(number)
```

This example uses a for loop to replace each element in the list with another value.

```
numbers = [1, 2, 3, 4, 5]
#fills each value in an existing numbers list using a for loop
for i in range(len(numbers)):
      numbers[i] = 3
print(numbers)
```

It is not essential that the **for** loop accesses every element. The loop example below will display every value up to the count of `batchSize`.

```
# displays every value up to the count of batchSize
batchSize = 5
for index in range(batchSize):
      print(numbers[index])
```

This example introduces the possibility of an "list index out of range" error. If the value of `batchSize` is larger than or equal to the length of the list, it will not be a valid index number for the list. In this case, the code will produce a run-time error.

This could be avoided by including an index checking condition before attempting to access `numbers[index]` e.g. `if index < len(numbers):` . . .

Using the derived value len

Notice that in the **while** loop the expression returning the length of the list is used for the loop limit rather than the expression `index < 10`. Using this **derived** value rather than the hardcoded value gives the code flexibility. If another number was added to the `numbers` list, its length would then be 11. The expression len(`numbers`) would still traverse the whole list correctly, whereas every loop using `index < 10` would need to be manually updated to 11. If this update is overlooked, the code will run, but it will stop 1 element short of the end of the list, so will likely produce wrong answers or output.

Summing every element of a list of integer values.

The code example below uses a for loop to add together every element stored in the list `numbers`.

```
numbers = [20, 36, 12, 24, 20, 48, 74, 353, 23, 98]
total = 0
for number in numbers:
      total = total + number
print("The sum of the numbers is " + str(total))
```

The total is, and needs to be, initialised outside of the **for** loop. (Why?)

Type this code into a Python window, save and run.

Try this

Adapt the code so that each number and the running total are displayed within the loop.

Expressions as indices

Any expression which produces an integer result within the valid range may be used as a list index.

```
composerList =["Mozart W.A.", 1756, 1791, "Bach J.S.", 1685, 1750]
index = 0
#process elements three at a time
while index + 2 < len(composerList): #notice the variation
  print(composerList[index], "born", composerList[index + 1], end = ', ')
  print("died", composerList[index + 2])
  index = index + 3 #updates to next composer's start index
```

The code above will produce the output:

```
Mozart W.A. born 1756, died 1791
Bach J.S. born 1685, died 1750
```

Try this:

Design a function which could collect data from the composer name, birth year and death year rather than have the values hardcoded in the program.

7.6 Practical

Problem 1:

The number 353 in the `numbers` list has been determined to be an error. It should have been 53. Find this value and replace it, then display the contents of the list to prove that it has been changed. (You cannot use a `for number in numbers` style of loop for this solution, as you will need to know the index of the integer to be replaced)

algorithm

start at first element
repeat
    check whether element matches 353
    if a match, change it to 53
    next element
until replacement made or end of data

Problem 2:

Write a program which finds the largest element in a list of float values. The algorithm suggests a solution. (Use the `numbers` list from the previous examples to test the implementation of the algorithm).

algorithm

first element is the largest
repeat
    if element is larger than largest
        largest = this element
    next element
until end of data

Problem 3:

Design and write a program which gets data for a list of float values from the user, then multiplies every element in the list by 3, overwriting the old values with the new values. Use a for loop to display all the values in the list twice, once before and once after the multiplication.

7.7 Lists as parameters

A list can be passed to a function as a parameter in the same way as any other variable.

The program below has a `displayRoll`function which takes a list as its input parameter. The main routine creates 2 different lists, and calls the `displayRoll` function once for each of them. The first time it is called, the local variable / input parameter `player` will be a copy of `player1`, the second time it will be a copy of `player2`.

```
#displays each element of the list parameter
def displayRoll(player):
    for eachDie in player:
        print(eachDie, end=' ')
    print()
#main routine
player1 = [2,6,4,5,5,1]
player2 = [4,2,4,5,4,6]
displayRoll(player1)
displayRoll(player2)
```

Type this code, save and run.

Try this

Add statements to the `displayRoll` function to calculate and display the total score for each player's roll.

Expected output:

```
2 6 4 5 5 1  Total is  23
4 2 4 5 4 6  Total is  25
```

7.8 Returning a list from a function

A list can be returned by a function, just like any other variable. In the example below, the function `makeList` returns a list.

```
#creates and returns a list of integers
def makeList(size):
  numbers = [0] * size
  for index in range (len(numbers)):
    numbers[index] = int(input("Enter an integer "))
  return numbers

#displays each value in a list of int
def printList(numberList):
  for number in numberList:
    print(str(number),end=' ')

#main routine
printList(makeList(5)) #see note
```

Note: The function `printList` wants to be sent a list as a parameter and the function `makeList` returns a list. They are made for each other. There is no need to go through the extra bother of saving the list in the main routine like this:

```
anotherList = makeList(5)
printList(anotherList)
```

7.9 Practical

Problem 4:

The main routine below shows a list (filled by initialiser list) which stores the height in centimetres of 11 children. The main routine goes on to call functions which have not been written.

```
#main routine
heights = [33, 45, 23, 43, 48, 32, 35, 46, 48, 39, 41]
printList(heights)
print("Average is " + str(findAverage(heights)))
print("Smallest is " + str(findSmallest(heights)))
print("Largest is " + str(findLargest(heights)))
print("Predicted heights after a year")
printList(nextYear(heights))
```

The points below describe the purpose of each unwritten function.

- Write a function which takes the list as an input parameter and displays each value in the list.
- Write a function which takes the list as an input parameter and returns the average height, formatted to 2 decimal places.
- Write a function which takes the list as an input parameter and returns the lowest height.
- Write a function which takes the list as an input parameter and returns the highest height.
- Write a function which takes the list as an input parameter and returns another list, representing the predicted heights after a year's growth. The expected growth for a year is 5 cm.

Expected output:

```
33 45 23 43 48 32 35 46 48 39 41
Average is 39.36
Smallest is 23
Largest is 48
Predicted heights after a year
38 50 28 48 53 37 40 51 53 44 46
```

Problem 5:

The highest house number in Queen St is 28. There are no flats or apartments. A travelling salesman wants to keep a record of which house numbers he has to visit. There is no house at number 2, 4, 7, 13, 15, 22, 23 or 24

Make a list of boolean values with indices which covers all the numbers to the highest house number. Display all the index numbers in this list except 0.

Use a while loop to request from the user a house number which doesn't exist. Change the element at that position from False (the default) to True for any house that does not exist. Exit this loop with a 0.

Print the numbers of the valid houses to be visited.

```
Possible house numbers
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28
Which house number doesn't exist? (0 to finish) 2

Valid house numbers
1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28
Which house number doesn't exist? (0 to finish) 4

Valid house numbers
1 3 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28
Which house number doesn't exist? (0 to finish) 7

Valid house numbers
1 3 5 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28

. . . and so on, removing 13, 15, 22, 23 and 24

Valid house numbers
1 3 5 6 8 9 10 11 12 14 16 17 18 19 20 21 25 26 27 28
Which house number doesn't exist? (0 to finish) 0

Valid house numbers
1 3 5 6 8 9 10 11 12 14 16 17 18 19 20 21 25 26 27 28
```

Extension task

Design and write a solution which allows the salesman to record the houses he has visited at the end of the day. (Ideally the data would have permanent storage in a file but this is outside the scope of this course. We will just hope the power to his computer stays on.)

Problem 6:

Rugby jerseys are numbered from 1 to 15. The table shows the player position attached to each jersey.

Number	Description
1	Loosehead prop
2	Hooker
3	Tighthead prop
4	Lock
5	Lock
6	Blindside flanker
7	Openside flanker
8	Number eight
9	Scrum-half
10	Fly-half
11	Left wing
12	Inside centre
13	Outside centre
14	Right wing
15	Full back

Write a program which puts the above info into a list. Put the coach in at position 0 so that the list index matches the jersey number.

Write a function which prints position 1 to 15 with the description.

Write a function which takes a number and a name as input parameter. The name should replace the description at that jersey number and report the fact.

Use the main routine to display the original data, make a substitution, then display the data again.

1 Loosehead prop
2 Hooker
3 Tighthead prop
4 Lock
5 Lock
6 Blindside flanker
7 Openside flanker
8 Number eight
9 Scrum-half
10 Fly-half
11 Left wing
12 Inside centre
13 Outside centre
14 Right wing
15 Full back

1 Ivan Boots
2 Hooker
3 Tighthead prop
4 Lock
5 Lock
6 Blindside flanker
7 Openside flanker
8 Number eight
9 Scrum-half
10 Fly-half
11 Left wing
12 Inside centre
13 Outside centre
14 Right wing
15 Full back

Replaced 1 with Ivan Boots

7.10 Random - optional

Random numbers are useful for games and for testing. They might be used for selecting a random element from a large list or for filling a list with random number values.

All the examples below assume that the line

```
import random
```

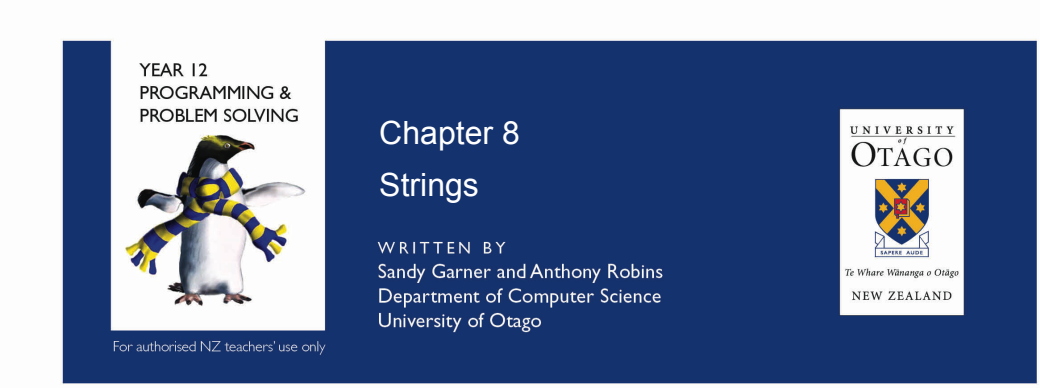
is written first, at the top of the file

```
random.randrange(20) # returns an integer between 0 and 19 inclusive
random.randrange(8) # returns an integer between 0 and 7 inclusive
random.randrange(8) + 3 # returns a value between 0 and 7 with an 'offset' of 3,
                          giving a value between 3 and 10
random.randrange(20) + 1 # returns a value between 0  and 19 with an 'offset'
                           of 1, giving a value between 1 and 20
random.randrange(3,10,2) # returns odd integers between 3 and 9 inclusive
random.randint(3,10) # returns an integer between 3 and 10 inclusive

low = 1
high = 3
rand = random.randint(low, high) # stores an integer between low and
                                   high inclusive

dieRoll = random.randrange(6) + 1 # integers from 1 to 6 inclusive

random.randrange(0, len(wordList))      # any index of wordList
random.randrange(len(wordList))         # any index of wordList
```

e.g.

```
wordList = ["apple","plum","pear"]
rand = random.randrange(0, len(wordList))
print(wordList[rand])
```

will print any one of the three elements of the list



8.0 Introduction

The data type `str` is a compound data type, which is made up of smaller parts i.e. characters. There are many useful ways of manipulating string objects, and they are not difficult to implement. This chapter will explore some of them.

8.1 String functions

Think of a string as a series of characters stored in a named sequence. Each character is fixed in its own position and has an index number. The numbers start at 0. The statements below assign `str` values to the variables `animal` and `colour`. These are then represented in the diagrams.

```
animal = "tree frog"
```

index	0	1	2	3	4	5	6	7	8
animal	t	r	e	e		f	r	o	g

```
colour = "red"
```

index	0	1	2
colour	r	e	d

A string can be of any length. Knowing the length of a string may be very useful in some circumstances. There is a function which returns the length of the string. This function, and any other in the string class, can be called using dot notation - the string's name is followed by a dot `'.'` then the method name e.g. for a string called `colour`, the length of the string can be displayed using the statement:

```
print(len(colour))  #displays length of the string specified
```

The length returned by the `len` function is the count of characters as humans count, starting at 1. In this case it would return 3. It is an integer, so its value can be used anywhere an integer might be used.

```
stringLength = len(colour)   #assigns the length to stringLength
```

Any string's index starts at 0 and finishes at `len - 1`.

The string declaration: `songLine = "Let it be"` is the basis for the diagram and examples below.

index	0	1	2	3	4	5	6	7	8
songLine	L	e	t		i	t		b	e

A space is a character like any other. Think of a character as any digit, letter or symbol which can be represented on the screen.

For the declaration `songLine = "Let it be"` the expression

`len(songLine)` will return the value _____.

selecting a character

`songLine[index]` will return the character at the specified index.

`letter = songLine[2]` will store `"t"` in the variable `letter`

`songLine[8]`will return the string value `"e"`

You cannot change a string so `songLine[1] = "o"` will not work.

slices

A slice is a substring of a string.To return a string which starts at the specified index and extends to the end of the string:

`songLine[1:]` will return the string `"et it be"`.

`songLine [7:]` will return the string `"be"`.

To return a string which starts at the specified index and extends up to but not including the second index:

`songLine[4:6]` will return the string "`it`".

`songLine[4:3]` will return the string `"Let"`.

To return a string which starts at the beginning and extends up to but not including the specified index:

`songLine[:5]` will return the string `"Let i"`.

Trying to access an index out of the valid range e.g. `songLine[17]` will produce an "IndexError: string index out of range" error when the code is run. In IDLE this appears in red. This is a run-time error.

The string class contains many methods which can be called. To see what might be available, in a Python shell, after the prompt (>>>) type `dir(str)`

To see how to use anything you already know the name of, after the prompt (>>>) type `help(str.replace)` etc.

replace

`replace(oldStr, newStr)` returns a string with all occurrences of `oldStr` replaced by `newStr` e.g.

`songLine.replace("e", "x")` will return "`Lxt it bx`"

`songLine.replace("Let", "x")` will return "`x it bx`"

upper

`upper()` returns a string made up of upper-case letters e.g.

`songLine.upper()` returns "LET IT BE"**lower**

`lower()` returns a string made up of lower-case letters e.g.

`songLine.lower()` returns "let it be"

in

substring `in` string returns True if the string contains the sequence of characters specified

`"Let" in songLine` returns `True`

__contains__

`__contains__(substring)` returns True if the string contains the sequence of characters specified (the "__" is 2 underscore characters) e.g.

`songLine.__contains__("Let")` returns `True`

`songLine.__contains__("let")` returns `False`

`songLine.upper().__contains__("LET")` returns `True`

find

`find(sub)` returns the index within this string of the **first** occurrence of the specified character/substring e.g.

`songLine.find("b")` will return the integer `7`.

`songLine.find("it")` will return the integer `4`.

`find(sub, fromIndex)` returns the index within this string of the first occurrence of the specified character/string, starting the search at the specified index.i.e. starts searching at fromIndex e.g.

`songLine.find("t", 3)` will return the integer `5`.

`songLine.find("e", 3)` will return the integer `8`.

If a character/substring cannot be found, the integer -1 will be returned.

`songLine.find("x")` will return the integer `-1`.

split

`split(char)` returns a list of strings split around the character specified e.g.

```
words = songLine.split(" ")
for word in words:
    print ("*" + word + "*")
```

will print out:

```
*Let*
*it*
*be*
```

Try this

For these examples, assume the declaration

```
drink = "Ginger ale"
```

Remember that the length of a string is the number of characters it contains (e.g. 10 for `drink`) but the index positions start at 0 and finish at `len() - 1`.

index	0	1	2	3	4	5	6	7	8	9
drink	G	i	n	g	e	r		a	l	e

What do each of the following expressions return?

`drink[0]` _____

`drink[len(drink) - 1]` _____

`drink[:7]` _____

`drink[3:len(drink) - 2]` _____

`drink.upper()` _____

`drink.lower()` _____

`drink.replace("e",":")` _____

`"ale" in drink` _____

`first_e = drink.find("e")` _____

If we want to find the second 'e', we need to start looking after the index of first 'e'.

`second_e = drink.find("e", 5)` _____

If the string stored in `drink` is changed from "Ginger Ale" to "green tea", the statement `second_e = drink.find("e", 5)` would no longer find the second 'e', it would find the third 'e'.

index	0	1	2	3	4	5	6	7	8
drink	G	r	e	e	n		t	e	a

The 5 is not flexible – it is "hardwired" and will not adapt if the string is changed. We can adapt the expression so that it finds the position of the second 'e' in the string drink in terms of the position of the first 'e':

```
second_e = drink.find("e", first_e + 1)
```

Comparing Strings

Compare the contents of strings using s1==s2.

Traversing Strings

The for loop is perfect for traversing a string, character by character.

```
quote = "Experience is the name everyone gives to their mistakes."

for i in range (0, len(quote)):
      print("character " + str(i) + ": " + quote[i])
```

The code above will produce the text:

```
character 0: E
character 1: x
character 2: p
etc. . . .
character 54: s
character 55: .
```

Or if you don't need the index numbers, just:

```
for char in quote:
    print(char)
```

Strings are immutable

You cannot change a string. You can build strings up by concatenation with other strings or char values e.g.

```
test = "I have a hundred duck"
c = "s"
test = test + c
print(test)        # will print I have a hundred ducks
```

What is happening behind the scenes is that `test = test + c` is creating a new string.

Note: If the code:

```
test = "hello"
print(test.upper())
```

is run in a Python program, it will print `HELLO`, but the string `test` remains unchanged and still refers to the value "`hello`". However after the statement:

```
test = test.upper()
```

the variable `test` refers to a different string.**8.3 Practical**

Problem 1:

Declare and instantiate a `list` of strings called `fish`

```
fish = ["flounder", "sole", "blue cod", "snapper", "terakihi",
"john dory", "red cod"]
```

a. print the first letter of each fish name on a new line

b. print the first 3 letters of each fish name on a new line

c. print the longest fish name

d. print out any fish name which is more than one word

e. print out any fish name which contains the word cod

Problem 2:

The user enters a sentence. Count the number of times the character `'e'` appears in the sentence.

Extension: write a function which takes two strings as input parameters, one representing a character to be found, the other representing the sentence in wich to search. The function should return the number of times the character appears in that string. The user enters the sentence as before.

Problem 3:

You have a `list` of words, let's say it is a shopping list. The user enters a word. The program checks whether the word is already in the list. If it is not, add the word to the end of the list. Repeat until the word "finish" is typed.

To make sure your check is accurate, remove any leading or trailing spaces and make sure the new item is in lowercase letters. If the string is empty, do not add it to the list.

Problem 4:

Copy or type a paragraph of text and store it as a string variable. Print it out as left-justified text with no more than 35 characters on a line.