
Learning Napier

# CalcScript

**Learning Napier CalcScript**
www.open-square.com

Open Square
2 Venture Road
Chilworth
Southampton
SO16 7NP
UK

## Contents

# 1 Introduction

## 1.1 Are you in the right place!

Welcome! We've designed this book as a primer about getting to grips with our Napier calculations service. You won't find it a great deal of use in any other context, I'm afraid. But, if that's what you were expecting, then please read on and enjoy…

Before we dive into the details about how to create wonderful Napier calculations, it might be useful to give a general understanding of how the service works and, more particularly, the distinctive approach taken by Napier that makes it the best way to implement the calculations which power your business. Boring? Please feel free to skip this section – it won't do you too much harm.

## 1.2 How Napier works


**Performing a calculation**

Napier accepts incoming calculation requests ❶ in a form you define (typically an xml message containing 'seed' data for a calculation).

This is unpacked by Napier ❷ which passes it to the calculation basis specified in the message.

The calculation basis ❸ contains the rules to perform the calculations required (including all necessary algorithms and tables).

The results are processed by Napier ❹ into a format you define,

and passed back to the calling application ❺.

Napier automatically keeps a record ❻ of the input and output messages it receives and creates, to allow for subsequent interrogation and reporting.

## 1.3 Creating a simple calculation

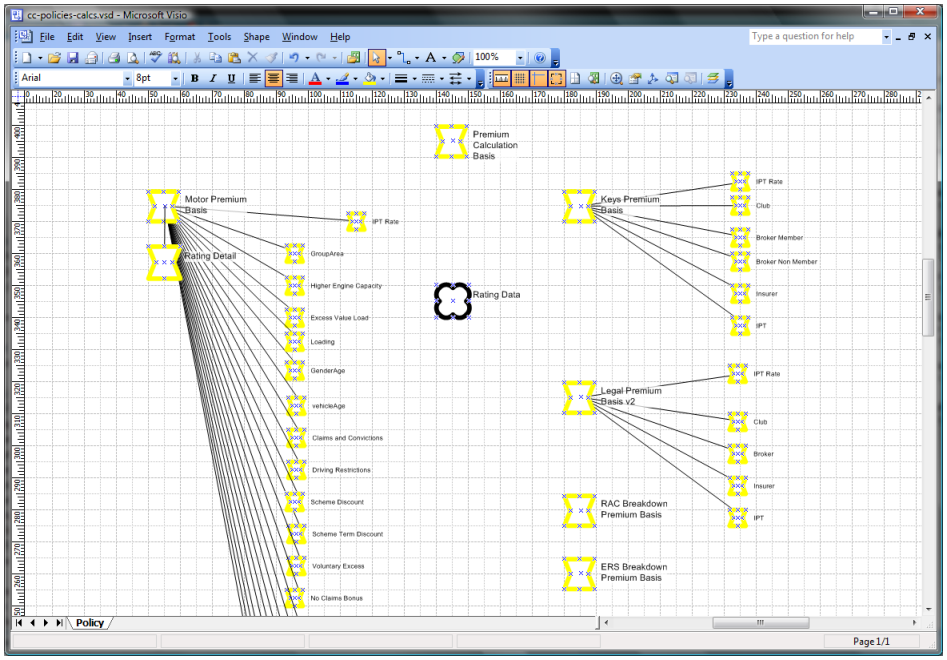
**Getting started – some prerequisites**

It is not difficult to dive in and get started creating your calculations and testing them out; but, before you begin you will need access to the following:

- a copy of Microsoft Visio with the Open Square Pact Add-In installed, and
- an account with a running Napier service,

Calculations are defined in Napier using an xml language. These files can be crafted by hand, but Visio provides an easy, more visual way, to proceed.

Using the Open Square Visio Add-In, the relevant symbols can be dragged into the model and configured. Right-clicking on a



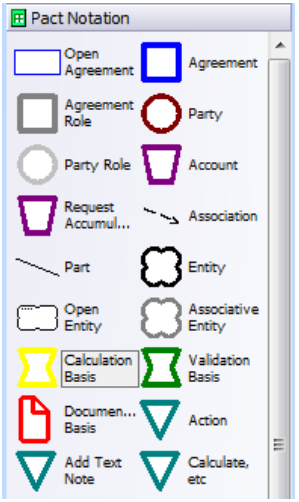
symbol then allows the requisite xml to be generated.

For the purposes of this document, a general working knowledge of Visio is assumed. (If you need help getting to grips with Visio, there are many resources around – try http://office.microsoft.com/en-gb/training/default.aspx , and search for "visio" to see Microsoft's own free videos.)

To operate your Napier account, you will also need the url of the **Napier Dashboard** (detailed below – see Chapter ???). This will allow you to login and install your calculations.

## 1.4 My first calculation

Open Visio and begin a new drawing. Drag the **Calculation Basis** symbol on to the drawing page. A calculation basis contains the rules which allow Napier to process a calculation. At its simplest, and for this first example, your calculation might comprise of just one of these yellow double-sigma symbols. Later, as in the example on the previous page, you will want to progress to more complex calculations which are made up of cascades of calculation bases – each basis defining a step in a calculation.

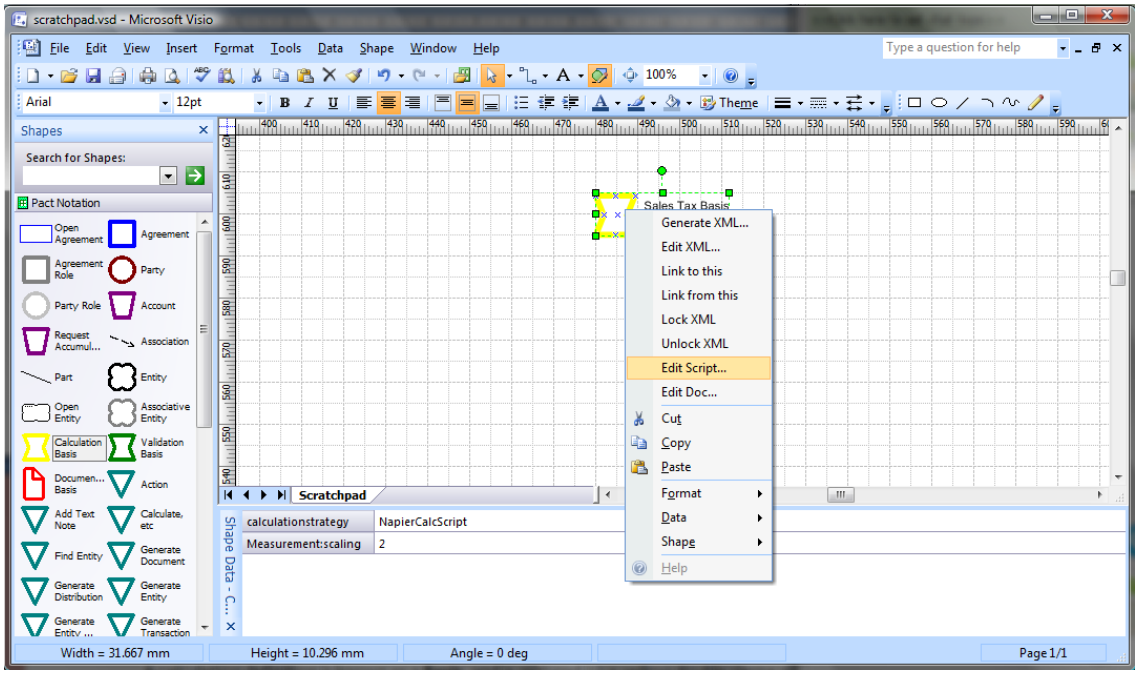



Each calculation basis has a **Calculation Strategy**. Think of the calculation strategy as being similar to a function in Excel (such as, SUM() or AVERAGE() ). Napier has a number of built in strategies, and these can be easily extended with new ones.

The most basic strategy is called "NapierCalcScript", it allows you to write out your calculation as a formula (a bit like typing "=A1+B2" into an Excel cell).

For this first example, we will produce a calculation which takes a supplied input value and multiplies this by a constant and returns the result. Sounds a bit simple? It is; but it's just the sort of calculation done all the time to add sales tax to a price. So let's give it a go…

Having dragged the yellow calculation basis into your drawing, simply type on the keyboard to replace the default name with something like “Sales Tax Basis”. Open up the Shape Data Window to see the properties of the symbol. You should see the calculation strategy set to NapierCalcScript.




To enter the formula for the calculation, right-click the basis symbol and select Edit Script… .

A simple text window opens into which we can type our formula.

If the sales tax is 15%, we could write something as simple as:

```
total = inputValue * 1.15
```

When run, this calculation would expect an input parameter named ‘inputValue’ and would return a result called ‘total’. Incidentally, the result will be rounded to two decimal places courtesy of the ‘scaling’ property set on the symbol (see illustration above).

Of course, for maintenance purposes the formula we have entered isn’t very clear. Instead we could write:

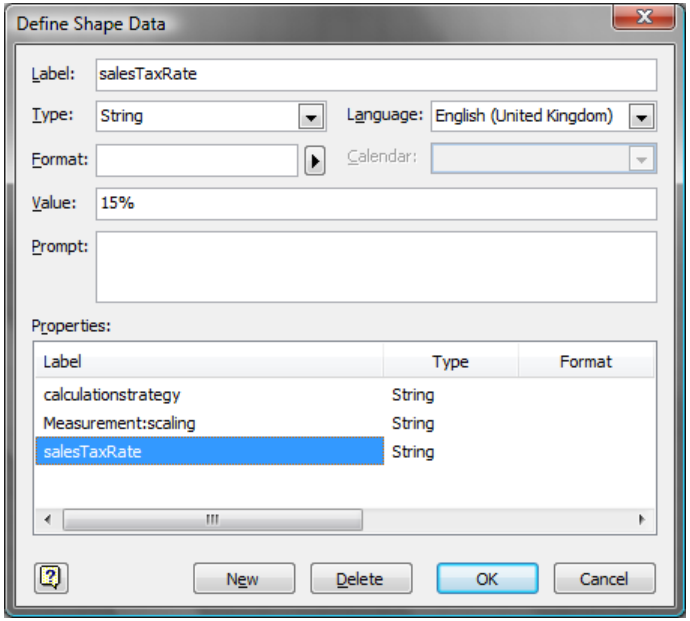
```
salesTaxRate = 1.15
total = inputValue * salesTaxRate
```

The first line defines a variable and its value, which is then used in the calculation proper. This may be more lines to write, but is far clearer to anyone else reading our work.

Better still might be to remove the 'salesTaxRate = 1.15' line from the script and define the variable as a property on the symbol. To do this, close the script editor, right click in the Shape Data Window (under 'Measurement:scaling 2' and select 'Define Shape Data…', then click 'New'. Add the SalesTaxRate property as follows shown here.

This approach has the advantage that others can see and amend the rate simply by inspecting the symbol, rather than opening up your script.

Ok, that's it, our first calculation written. Now, let's give it a go…

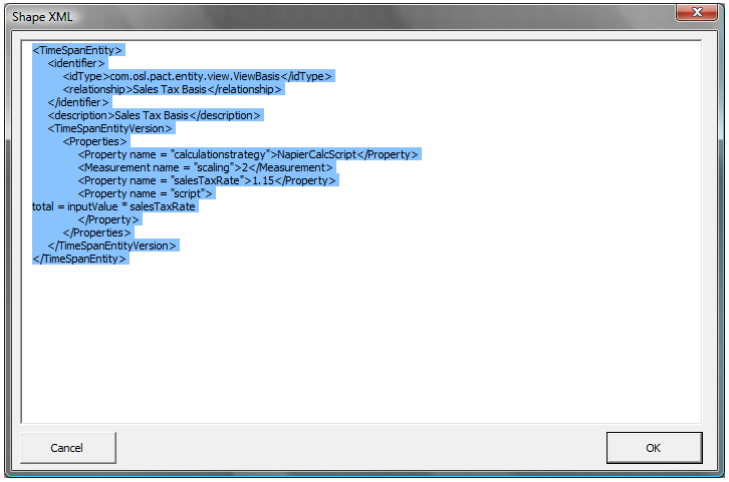



## 1.5 Loading our calculation

To use your newly created template, you need to upload it to Napier. Simply, follow these steps:

- Login to your Napier Dashboard with your username and password
- Go to the **Calculations Rules** tab of the Dashboard
- Go to the **Upload a resource** section of the page, with Type set as **Basis**, enter a Name and Effective period (be sure it covers the current day!)

- Switch to the Visio diagram and right click the yellow basis symbol to select 'Generate xml…' – you should get a window like that below. Copy all the text (Ctrl-A, Ctrl-C).




- Switch back to the Napier Dashboard, and paste the xml into the **XML** box (Ctrl-V).
- Finally, click the **Upload** button. Done!

Your calculation basis should now appear in the Resources view on that page.

## 1.6 More about calculation bases

**Reserved properties**

We have already seen two properties on the default calculation basis symbol that have a special meaning;

| | |
|---|---|
| **CalculationStrategy** | Tells Napier how to handle this calculation step engine. By default, set to 'NapierCalcScript' which allows a freeform algorithm script to be used. |
| **Scaling** | The number of decimal places to be retained in the result. |

Other properties with specific purposes are:

| | |
|---|---|
| **propertynames** | A comma-separated list of properties, telling Napier the names of variables to expect in an input message. |
| **partnames** | A comma-separated list of parts, telling Napier the structure of the input message. |
| **assocnames** | A comma-separated list of associated entities (based on relationship:assocrelationship, for example 'Party Role:insured'. |
| **includebalances** | Set to 'true' if account balances are needed. |
| **includepostings** | Set to 'true' if account postings are needed (in which case the invocation must include a start TimePoint, or will default to entity start time). |
| **Quantifier** | The name, if any, of a special Measurement property on the base entity. |
| **unit** | The unit to be attached to the result, e.g. a currency code. If the entity has a property designated as a 'quantifier' then the currency will be defaulted from that property. |
| **updateproperties** | A comma-separated list of properties which may be updated with the calculation result (depending whether this Basis is referenced in an active or passive context). |

### NapierCalcScript

This is the default strategy, as we have seen. This allows you to formulate an algorithm of arbitrary complexity by writing it in the Napier CalcScript language. For more on how to use this language, see Section ???.

# 2 Getting to grips with Napier CalcScript

## 2.1 Some basics – variables, numbers and strings

Napier CalcScript is based on the computer programming language Python. Here is a quick introduction to the main features based upon the work of Swaroop C H (www.swaroopch.com) and made available by him under a Creative Commons Attribution Share-Alike license (that means you are also free to use our adaption of his work as it appears in this chapter).

**Numbers**

Numbers are of three types - integers, floating point and complex numbers.

- An example of an integer is 2 which is just a whole number.
- Examples of floating point numbers (or floats for short) are 3.23 and 52.3E-4. The E notation indicates powers of 10. In this case, 52.3E-4 means 52.3 * 10-4.
- Examples of complex numbers are (-5+4j) and (2.3 - 4.6j)

(Note for experienced programmers - there is no separate 'long int' type. The default integer type can be any large value.)

**Strings**

A string is a sequence of characters. The characters can be in English or any other language that is supported in the Unicode standard, which means almost any language in the world.

You can specify strings using single or double quotes such as 'Quote me on this' or "What's your name?". All white space i.e. spaces and tabs are preserved as-is.

You can specify multi-line strings using triple quotes - (""" or '''). You can use single quotes and double quotes freely within the triple quotes. This can be useful, particularly for enclosing multiple lines of text.

(Note for experienced programmers – you can escape reserved characters by using a backslash \ . This means that if you actually want to use a backslash, then you need to double it, so \\ . Also \n inserts a new line, and \t a tab. You can create 'raw' strings where no processing or escaping is performed, this is done by prefixing an r or R to the string, thus: r"Newlines are indicated by \n".)

**The format Method**

Sometimes we may want to construct strings from other information. This is where the format() method is useful. Consider this code fragment…

```
age = 25
name = 'Swaroop'
output = '{0} is {1} years old'.format(name, age)
```

The result will be a text string in the variable output which equals 'Swaroop is 25 years old'.

Observe that {0} corresponds to the variable name which is the first argument to the format method. Similarly, the second specification is {1} corresponding to age which is the second argument to the format method.

This approach has advantages over using string concatenations (e.g. name + ' is ' + str(age) + ' years old' ) as it is less error-prone and the conversion to string is handled automatically by the format method instead of requiring an explicit conversion.

**Variables**

Variables are exactly what they mean - their value can vary (i.e. you can store anything using a variable). Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names. There are some rules you have to follow when selecting a name:

- The first character must be a letter of the alphabet or an underscore ('_').
- The rest of the name can consist of letters, underscores ('_') or digits (0-9).
- Names are case-sensitive. For example, myname and myName are not the same.

Variables can hold values of different types called data types. The basic types are numbers and strings, as discussed above.

**A simple program**

```
i = 5
i = i + 1
output = i
```

Here we assign the number value 5 to a variable named i, then we add 1 to the value stored in i and store it back. We then move the result to a variable named 'output', hopefully with the value 6.

Implicitly, Python encourages the use of a single statement per line which makes code more readable. If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using a semicolon (;) which indicates the end of a logical line/statement. However, I strongly recommend that you stick to writing a single logical line in a single physical line only. Use more than one physical line for a single logical line only if the logical line is really long. The idea is to avoid the semicolon as far as possible since it leads to more readable code. In fact, I have never used or even seen a semicolon in a Python program.

**Indentation**

Whitespace is important. Actually, whitespace at the beginning of the line is important. This is called indentation. Leading whitespace (spaces and tabs) at the beginning of a line is used to determine the indentation level of the line, which in turn is used to determine the grouping of statements. This means that statements which go together must have the same indentation. Each such set of statements is called a block. We will see examples of how blocks are important below.

Incorrect indentation can give rise to errors!

**How to indent**

Do not use a mixture of tabs and spaces for the indentation as it does not work across different platforms properly. I strongly recommend that you use a single tab or four spaces for each indentation level. Choose either of these two indentation styles. More importantly, choose one and use it consistently, i.e. use that indentation style only.

## 2.2 Operators and expressions

Most statements (logical lines) that you write will contain **expressions**. A simple example of an expression is `2 + 3`. An expression can be broken down into operators and operands.

*Operators* are functionality that do something and can be represented by symbols such as `+` or by special keywords. Operators require some data to operate on and such data is called *operands*. In this case, `2` and `3` are the operands.

We will briefly take a look at the operators and their usage:

| Operator | Name | Explanation | Examples |
|---|---|---|---|
| + | Plus | Adds the two objects | `3 + 5` gives `8`. `'a' + 'b'` gives `'ab'`. |
| - | Minus | Either gives a negative number or gives the subtraction of one number from the other | `-5.2` gives a negative number. `50 - 24` gives `26`. |
| * | Multiply | Gives the multiplication of the two numbers or returns the string repeated that many times. | `2 * 3` gives `6`. `'la' * 3` gives `'lalala'`. |

| | | | |
|---|---|---|---|
| ** | Power | Returns x to the power of y | `3 ** 4` gives `81` (i.e. `3 * 3 * 3 * 3`) |
| / | Divide | Divide x by y | `4 / 3` gives `1.3333333333333333`. |
| // | Floor Division | Returns the floor of the quotient | `4 // 3` gives `1`. |
| % | Modulo | Returns the remainder of the division | `8 % 3` gives `2`. `-25.5 % 2.25` gives `1.5`. |
| << | Left Shift | Shifts the bits of the number to the left by the number of bits specified. (Each number is represented in memory by bits or binary digits i.e. 0 and 1) | `2 << 2` gives `8`. `2` is represented by `10` in bits. Left shifting by 2 bits gives `1000` which represents the decimal `8`. |
| >> | Right Shift | Shifts the bits of the number to the right by the number of bits specified. | `11 >> 1` gives `5`. `11` is represented in bits by `1011` which when right shifted by 1 bit gives `101` which is the decimal `5`. |
| & | Bitwise AND | Bitwise AND of the numbers | `5 & 3` gives `1`. |
| \| | Bit-wise OR | Bitwise OR of the numbers | `5 \| 3` gives `7` |
| ^ | Bit-wise XOR | Bitwise XOR of the numbers | `5 ^ 3` gives `6` |
| ~ | Bit-wise invert | The bit-wise inversion of x is -(x+1) | `~5` gives `-6`. |

| | | | |
|---|---|---|---|
| < | Less Than | Returns whether x is less than y. All comparison operators return `True` or `False`. Note the capitalization of these names. | `5 < 3` gives `False` and `3 < 5` gives `True`. Comparisons can be chained arbitrarily: `3 < 5 < 7` gives `True`. |
| > | Greater Than | Returns whether x is greater than y | `5 > 3` returns `True`. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns `False`. |
| <= | Less Than or Equal To | Returns whether x is less than or equal to y | `x = 3; y = 6; x <= y` returns `True`. |
| >= | Greater Than or Equal To | Returns whether x is greater than or equal to y | `x = 4; y = 3; x >= 3` returns `True`. |
| == | Equal To | Compares if the objects are equal | `x = 2; y = 2; x == y` returns `True`.<br>`x = 'str'; y = 'stR'; x == y` returns `False`.<br>`x = 'str'; y = 'str'; x == y` returns `True`. |
| != | Not Equal To | Compares if the objects are not equal | `x = 2; y = 3; x != y` returns `True`. |
| not | Boolean NOT | If x is `True`, it returns `False`. If x is `False`, it returns `True`. | `x = True; not x` returns `False`. |
| and | Boolean AND | `x and y` returns `False` if x is `False`, else it returns evaluation of y | `x = False; y = True; x and y` returns `False` since x is False. In this case, Python will not evaluate y since it knows that the left hand side of the 'and' expression is `False` which |

| | | | |
|---|---|---|---|
| | | | implies that the whole expression will be `False` irrespective of the other values. This is called short-circuit evaluation. |
| or | Boolean OR | If x is `True`, it returns True, else it returns evaluation of y | `x = True; y = False; x or y` returns `True`. Short-circuit evaluation applies here as well. |

**Shortcut for math operation and assignment**

It is common to run a math operation on a variable and then assign the result of the operation back to the variable, hence there is a shortcut for such expressions:

You can write:

```
a = 2; a = a * 3
```
as:

```
a = 2; a *= 3
```

**Evaluation Order**

If you had an expression such as `2 + 3 * 4`, is the addition done first or the multiplication? Our high school maths tells us that the multiplication should be done first. This means that the multiplication operator has higher precedence than the addition operator.

The following table gives the precedence table for Python, from the lowest precedence (least binding) to the highest precedence (most binding). This means that in a given expression, Python will first evaluate the operators and expressions lower in the table before the ones listed higher in the table. (The table is taken from the Python reference manual, and is provided for the sake of completeness. It is far better to use parentheses to group operators and operands appropriately in order to explicitly specify the precedence. This makes the program more readable - see **Changing the order of evaluation** below for details.)

| Operator | Description |
|---|---|
| lambda | Lambda Expression |
| or | Boolean OR |
| and | Boolean AND |
| not x | Boolean NOT |
| in, not in | Membership tests |
| is, is not | Identity tests |
| <, <=, >, >=, !=, == | Comparisons |
| \| | Bitwise OR |
| ^ | Bitwise XOR |
| & | Bitwise AND |
| <<, >> | Shifts |
| +, - | Addition and subtraction |

| Operator | Description |
|---|---|
| *, /, //, % | Multiplication, Division, Floor Division and Remainder |
| +x, -x | Positive, Negative |
| ~x | Bitwise NOT |
| ** | Exponentiation |
| x.attribute | Attribute reference |
| x[index] | Subscription |
| x[index1:index2] | Slicing |
| f(arguments ...) | Function call |
| (expressions, ...) | Binding or tuple display |
| [expressions, ...] | List display |
| {key:datum, ...} | Dictionary display |

Operators with the *same precedence* are listed in the same row in the above table. For example, `+` and `-` have the same precedence.

**Changing the order of evaluation**

To make the expressions more readable, we can use parentheses. For example, `2 + (3 * 4)` is definitely easier to understand than `2 + 3 * 4` which requires knowledge of the operator precedences. As with everything else, the parentheses should be used reasonably (do not overdo it) and should not be redundant (as in `2 + (3 + 4)`). There is an

additional advantage to using parentheses - it helps us to change the order of evaluation. For example, if you want addition to be evaluated before multiplication in an expression, then you can write something like `(2 + 3) * 4`.

**Associativity**

Operators are usually associated from left to right i.e. operators with same precedence are evaluated in a left to right manner. For example, `2 + 3 + 4` is evaluated as `(2 + 3) + 4`. Some operators like assignment operators have right to left associativity i.e. `a = b = c` is treated as `a = (b = c)`.

## 2.3 Controlling flow

In the programs we have seen till now, there has always been a series of statements executed in the same order. What if you wanted to change the flow of how it works? For example, you want a calculation to take some decisions and do different things depending on different situations such as the time of the day?

As you might have guessed, this is achieved using control flow statements. There are three control flow statements in Python - `if`, `for` and `while`.

**The if statement**

The `if` statement is used to check a condition and *if* the condition is true, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*). The *else* clause is optional.

Example:

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    print('Congratulations, you guessed it.') # New block starts here
    print('(but you do not win any prizes!)') # New block ends here
```

```
elif guess < number:
    print('No, it is a little higher than that') # Another block
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guess > number to reach here

print('Done')
# This last statement is always executed, after the if statement is executed
```

Output:

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
Done

$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done

$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

How It Works:

In this program, we take guesses from the user and check if it is the number that we have. We set the variable `number` to any integer we want, say `23`. Then, we take the user's guess using the `input()` function. Functions are just reusable pieces of programs. We'll read more about them in the next chapter.

We supply a string to the built-in `input` function which prints it to the screen and waits for input from the user. Once we enter something and press `enter` key, the `input()` function returns what we entered, as a string. We then convert this string to an integer using `int` and then store it in the variable `guess`. Actually, the `int` is a class but all you need to know right now is that you can use it to convert a string to an integer (assuming the string contains a valid integer in the text).

Next, we compare the guess of the user with the number we have chosen. If they are equal, we print a success message. Notice that we use indentation levels to tell Python which statements belong to which block. This is why indentation is so important in Python. I hope you are sticking to the "consistent indentation" rule. Are you?

Notice how the `if` statement contains a colon at the end - we are indicating to Python that a block of statements follows.

Then, we check if the guess is less than the number, and if so, we inform the user to guess a little higher than that. What we have used here is the `elif` clause which actually combines two related `if else-if else` statements into one combined `if-elif-else` statement. This makes the program easier and reduces the amount of indentation required.

The `elif` and `else` statements must also have a colon at the end of the logical line followed by their corresponding block of statements (with proper indentation, of course)

You can have another `if` statement inside the if-block of an `if` statement and so on - this is called a nested `if` statement.

Remember that the `elif` and `else` parts are optional. A minimal valid `if` statement is:

```
if True:
    print('Yes, it is true')
```

After Python has finished executing the complete `if` statement along with the associated `elif` and `else` clauses, it moves on to the next statement in the block containing the `if` statement. In this case, it is the main block where execution of the program starts and the next statement is the `print('Done')` statement. After this, Python sees the ends of the program and simply finishes up.

Although this is a very simple program, I have been pointing out a lot of things that you should notice even in this simple program. All these are pretty straightforward (and surprisingly simple for those of you from C/C++ backgrounds) and requires you to become aware of all these initially, but after that, you will become comfortable with it and it'll feel 'natural' to you.

Note for C/C++ Programmers

> There is no `switch` statement in Python. You can use an `if..elif..else` statement to do the same thing (and in some cases, use a dictionary to do it quickly)

**The while Statement**

The `while` statement allows you to repeatedly execute a block of statements as long as a condition is true. A `while` statement is an example of what is called a *looping* statement. A `while` statement can have an optional `else` clause.

Example:

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        running = False # this causes the while loop to stop
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

Output:

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

How It Works:

In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly run the program for each guess, as we have done in the previous section. This aptly demonstrates the use of the `while` statement.

We move the `input` and `if` statements to inside the `while` loop and set the variable `running` to `True` before the while loop. First, we check if the variable `running` is `True` and then proceed to execute the corresponding *while-block*. After this block is executed, the condition is again checked which in this case is the `running` variable. If it is true, we execute the while-block again, else we continue to execute the optional else-block and then continue to the next statement.

The `else` block is executed when the `while` loop condition becomes `False` - this may even be the first time that the condition is checked. If there is an `else` clause for a `while` loop, it is always executed unless you break out of the loop with a `break` statement.

The `True` and `False` are called Boolean types and you can consider them to be equivalent to the value `1` and `0` respectively.

Note for C/C++ Programmers

> Remember that you can have an `else` clause for the `while` loop.

**The for loop**

The `for..in` statement is another looping statement which *iterates* over a sequence of objects i.e. go through each item in a sequence. We will see more about sequences in detail in later chapters. What you need to know right now is that a sequence is just an ordered collection of items.

Example:

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

Output:

```
$ python for.py
1
2
3
4
The for loop is over
```

How It Works:

In this program, we are printing a *sequence* of numbers. We generate this sequence of numbers using the built-in `range` function.

What we do here is supply it two numbers and `range` returns a sequence of numbers starting from the first number and up to the second number. For example, `range(1,5)` gives the sequence `[1, 2, 3, 4]`. By default, `range` takes a step count of 1. If we supply a third number to `range`, then that becomes the step count. For example, `range(1,5,2)` gives `[1,3]`. Remember that the range extends *up to* the second number i.e. it does **not** include the second number.

The `for` loop then iterates over this range - `for i in range(1,5)` is equivalent to `for i in [1, 2, 3, 4]` which is like assigning each number (or object) in the sequence to i, one at a time, and then executing the block of statements for each value of `i`. In this case, we just print the value in the block of statements.

Remember that the `else` part is optional. When included, it is always executed once after the `for` loop is over unless a break statement is encountered.

Remember that the `for..in` loop works for any sequence. Here, we have a list of numbers generated by the built-in `range` function, but in general we can use any kind of sequence of any kind of objects! We will explore this idea in detail in later chapters.

Note for C/C++/Java/C# Programmers

> The Python `for` loop is radically different from the C/C++ `for` loop. C# programmers will note that the `for` loop in Python is similar to the `foreach` loop in C#. Java programmers will note that the same is similar to `for (int i : IntArray)` in Java 1.5 .
>
> In C/C++, if you want to write `for (int i = 0; i < 5; i++)`, then in Python you write just `for i in range(0,5)`. As you can see, the `for` loop is simpler, more expressive and less error prone in Python.

**The break Statement**

The `break` statement is used to *break* out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become `False` or the sequence of items has been completely iterated over.

An important note is that if you *break* out of a `for` or `while` loop, any corresponding loop `else` block is **not** executed.

Example:

```
while True:
    s = (input('Enter something : '))
    if s == 'quit':
```

```
        break
    print('Length of the string is', len(s))
print('Done')
```

Output:

```
$ python break.py
Enter something : Programming is fun
Length of the string is 18
Enter something : When the work is done
Length of the string is 21
Enter something : if you wanna make your work also fun:
Length of the string is 37
Enter something :       use Python!
Length of the string is 12
Enter something : quit
Done
```

How It Works:

In this program, we repeatedly take the user's input and print the length of each input each time. We are providing a special condition to stop the program by checking if the user input is `'quit'`. We stop the program by *breaking* out of the loop and reach the end of the program.

The length of the input string can be found out using the built-in `len` function.

Remember that the `break` statement can be used with the `for` loop as well.

**The continue Statement**

The `continue` statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

Example:

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
        continue
    print('Input is of sufficient length')
    # Do other kinds of processing here...
```

Output:

```
$ python test.py
Enter something : a
Too small
Enter something : 12
Too small
Enter something : abc
Input is of sufficient length
Enter something : quit
```

How It Works:

In this program, we accept input from the user, but we process them only if they are at least 3 characters long. So, we use the built-in `len` function to get the length and if the length is less than 3, we skip the rest of the statements in the block by using the `continue` statement. Otherwise, the rest of the statements in the loop are executed and we can do any kind of processing we want to do here.

Note that the `continue` statement works with the `for` loop as well.

## 2.4 Using Excel spreadsheets in CalcScript

A Calculation Basis that uses the NapierCalcScript calculation strategy can make use of the ExcelHelper class to obtain values from an Excel Spreadsheet containing, typically, a table of values to be looked up.

The usage of this feature can be illustrated by this example:

```
# create a new Excel Helper
eh = ExcelHelper()

# prepare the key items as a java list
keylist = list([gender, smoker, age])

sheet = "mortality"
ratecolumn = "Value"

#invoke the lookup, and evaluate the (string) result
rate = eval(eh.getValue("C:\\location\of\Excel\file\filename.xls", sheet, keylist,
ratecolumn))
```

# 3 Next steps