

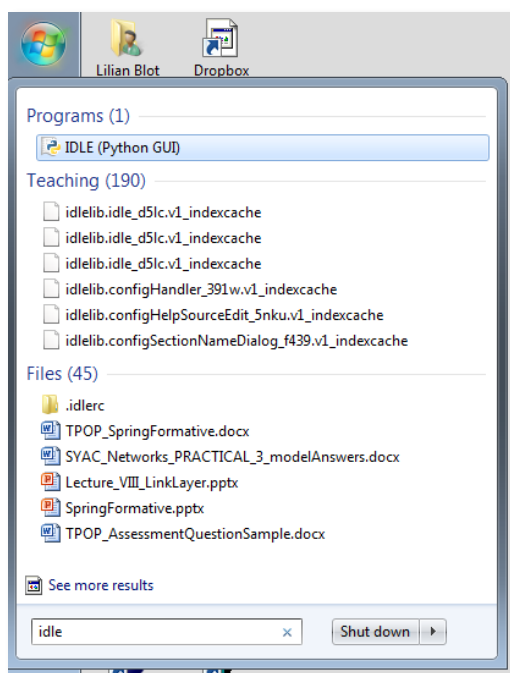
SOFTWARE 1 PRACTICAL

ELEMENTARY PROGRAMMING

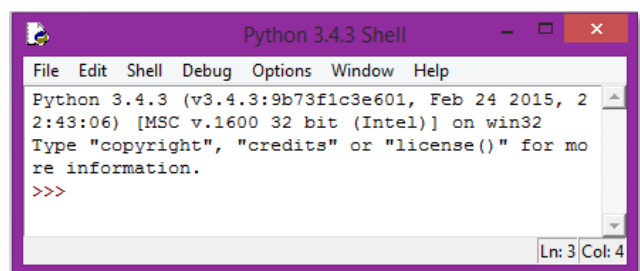
WEEK 1

Getting started

The first thing we need to do is opening a Python shell. Go to the Windows start button and type idle (see Figure 1a). A new window (shell) will open as shown in Figure 1b (Note, program version may vary). We will use this shell to type our code for the time being. Toward the end of the lecture we will be saving our code in a file.



(a)



(b)

Figure 1

First lines of code

A Python shell could be considered as a big calculator where we can do calculation using some mathematical expression. The basic element of an expression is a value. In Figure 2, the first line of code represents a value, which is a number and has value 1. We can also use known operators such as addition, multiplication, etc. Such combination of operators and operand is known as an expression. Once you have typed an expression and pressed the return key, the expression is evaluated and the result is printed in blue on the line below.

As you know mathematical operators have rules of precedence, it is the same here. The use of parenthesis can be used to counteract the rules of precedence between operators as shown in Figure 2, lines of code 4 and 5.

```
>>> 4/2
2.0
>>>
>>> 4//2
2
>>> 3//2
1
>>>
```

Figure 3

When considering the division operator `//` something strange happen. `3//2` gives 1. This is called the “integer division” operator. What are the results of `-3/2`, `-3/-2` and `-3.0/-2`?

```
>>> 1
1
>>> 1 + 2
3
>>> 3 * 9
27
>>> 2 * 3 + 4
10
>>> 2 * (3 + 4)
14
>>> -1 - 3
-4
>>> 1 + (-2)
-1
>>> 1 + -2
-1
>>> -2 * -4
8
```

Figure 2

Even old calculators (e.g. when I was still at school) had a way to memorised values that can be reused later. We have the same principle in Python. Such objects are called variables. In Figure 4 we declare a variable `x` and assign the value 2 to it. The `=` sign is called the assignment operator, `x` is the name of the variable, and 2 is the value assigned to `x`. Every time we use `x` it is the same (almost) as using the value 2. We can use `x` in any expressions, we can have more than one variable (here `x` and `y`); we can also use several variables in the same expression as shown in Figure 4. The sixth line of code shows how we can change the value in a variable. From now on, `x` has the value 5 in memory and do not remember having the value 2.

```
>>> x = 2
>>> x
2
>>> x + 3
5
>>> 2 * x
4
>>> x + x
4
>>> x = 5
>>> x
5
>>> x + 3
8
>>> y = 4
>>> x + y
9
```

Figure 4

Using `x` and `y` as name for variables is allowed, however it is not very meaningful. In Figure 5 the variables name are very explicit and therefore it is not difficult to understand what the aim

```
>>> number_cakes = 4
>>> cake_price = 2.50
>>> bill = cake_price * number_cakes
>>> bill
10.0
>>>
```

Figure 5

of the program is. You are strongly encouraged to do the same thing. You **MUST** read the Style Guide for Python Code available at:

<http://www.python.org/dev/peps/pep-0008/>

My first error (and probably not the last one)

What happen if I misspell the name of a variable? I will get an error as we are asking the interpreter to use something it does not know. The interpreter will promptly show you its discontentment in red as shown in Figure 6 .

```
>>> number_cakes = 4
>>> cake_price = 2.50
>>> bill = cake_price * number_cake

Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    bill = cake_price * number_cake
NameError: name 'number_cake' is not defined
>>>
```

Figure 6

It is very important for you to recognise the type of error and try to fix it. Python interpreter try its best to help you in this task. Here it points to the line containing an error `<bill = ...>` followed by the type of error `<NameError:...>`.

In our case it tells us that `number_cake` is not defined. Indeed we misspelled the name of the variable forgetting the 's' at the end. When you encounter an error that you are not familiar with I encourage you to search an explanation online. If it proves unsuccessful, ask myself or one of the PTAs.

Another data type

So far we have seen two types of numbers, does Python have any other type of values? The answer to that question is yes, Python does have quite a few more types. The next one we are going to see represents a series of character to form words or sentences for example. Values of

```
>>> 'this is a succession of characters.'
'this is a succession of characters.'
>>> "this is a succession of characters."
'this is a succession of characters.'
>>> type('this is a succession of characters.')
<class 'str'>
>>> |
```

Figure 7

this type are called string. Figure 7 shows a string representing a sentence. Python has two ways of representing a string, using single quotes or double quotes. In the same way as for numbers, string can be assigned to variables (Figure 8) and we can use the command `print` to print the string

value on the console. For example the statement `print words` display the content of the variable `words` on the console.

```
>>> print('this is a succession of characters.')
this is a succession of characters.
>>> words = 'this is a succession of characters.'
>>> words
'this is a succession of characters.'
>>> print(words)
this is a succession of characters.
>>> sentence = 'This is another sentence.'
>>> print(words + sentence)
this is a succession of characters.This is another sentence.
>>> print(words, sentence)
this is a succession of characters. This is another sentence.
>>>
```

Figure 8

To concatenate two strings, e.g. build one string from two substrings, we can use the addition operator `+`. For example:

```
longString = words+sentence
```

We can also print several strings in the same statement by separating them with a comma. Note that by doing so a space will be added between the two strings.

More errors

```
>>> another_sentence = "Black Knight: All right, we'll call it a draw."
SyntaxError: EOL while scanning string literal
>>>
>>> another_sentence = 'Black Knight: All right, we'll call it a draw.'
SyntaxError: invalid syntax
>>> another_sentence = "Black Knight: All right, we'll call it a draw."
>>>
```

Figure 9

You should be careful when declaring strings that you are using the same delimiter at the start and end of the string (first error in Figure 9). What about the second error? How can I do if I want to use single quote or double quotes in a string? Look on the web to find the answer.

First attempt at a small program

Let's try to create a small program that compute a bill for a cake shop. We would like to get the number of cakes, the price of a single cake and then print the total price of the bill. Figure 10 shows such a small program where the number of cake is four.

```
>>> number_cakes = 4
>>> cake_price = 2.50
>>> bill = cake_price * number_cakes
>>> print('The price of', number_cakes, 'cake(s) is', bill, 'pounds.')
The price of 4 cake(s) is 10.0 pounds.
>>>
```

Figure 10

Now if I want to compute a bill for seven cakes I need to rewrite the code as shown in Figure 11. As you can see the only change between the two scripts is the assignment of the value 7 to the variable `number_cakes`.

```
>>> number_cakes = 7
>>> cake_price = 2.50
>>> bill = cake_price * number_cakes
>>> print('The price of', number_cakes, 'cake(s) is', bill, 'pounds.')
The price of 7 cake(s) is 17.5 pounds.
>>>
```

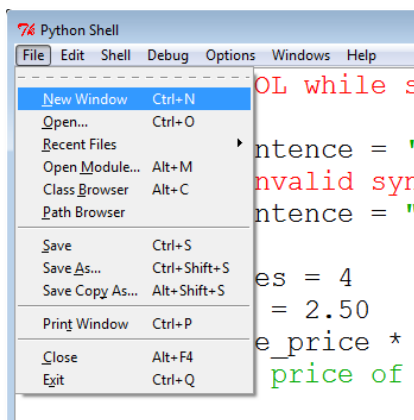
Figure 11

And if I want to do that again I will have to retype everything one more time. This is clearly not efficient. What can I do to change that? This is the subject of the next section.

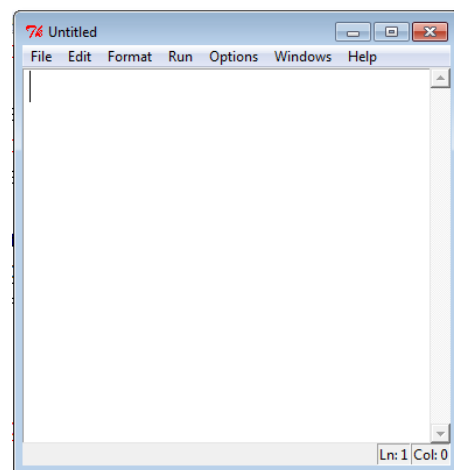
How to create a module?

To solve the problem of rewriting the same code over and over, we will use a file to store the lines of codes. In Python it is known as a Module. We will talk a little bit more about module later during the year. It is an important concept and we will need more time to discuss it in depth. So for the moment just consider a module as a place to store you code so it can be reused/run multiple times.

From the Python shell select the file menu and click on the <New Window> item (Figure 13a). A new window will open that is different from the Python shell (Figure 13b). This is the place where we will be writing our code.



(a)



(b)

Figure 13

Rewrite the code in the window as shown in Figure 12. Note that some part of the code is highlighted with different colours. Before we can run the code, we need to save the file. Select save as from the menu and type the name of the module, it must start with a letter and by convention contains only lower case letters, numbers and underscores. You must **manually** add the .py extension to keep the colour highlighting.

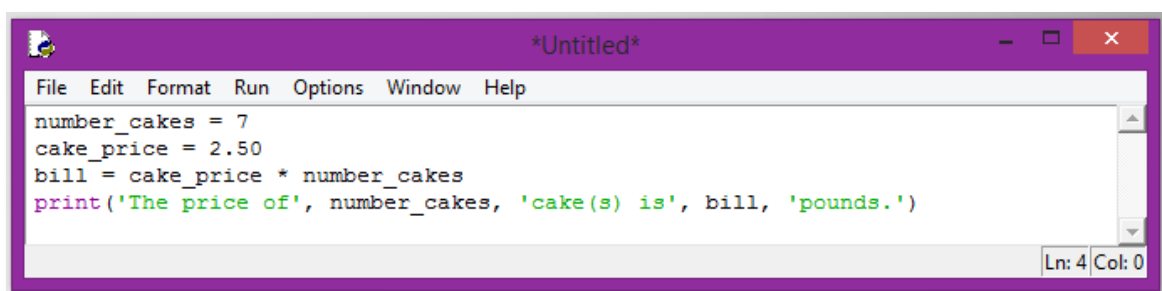


Figure 12

Trouble shooting: if your code appears in black only, this means you did not add the extension .py in your filename. Repeat the save as operation with the correct extension to solve the problem.

Running my first program

Once we have written our code, the next step is to run it. From the module window containing our code, select Run from the menu and click on the <Run Module> item. Alternatively press the function key F5 (Figure 14). You must save your changes before running the program in order to run the latest version. Python will remind you if you have forgotten.

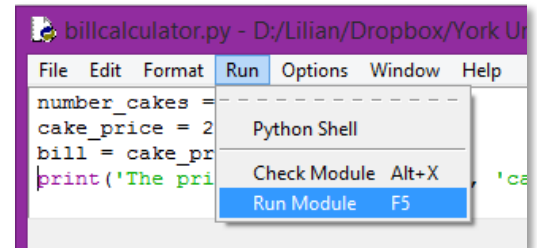


Figure 14

The result of the execution will appear in the Python shell as shown in Figure 15.

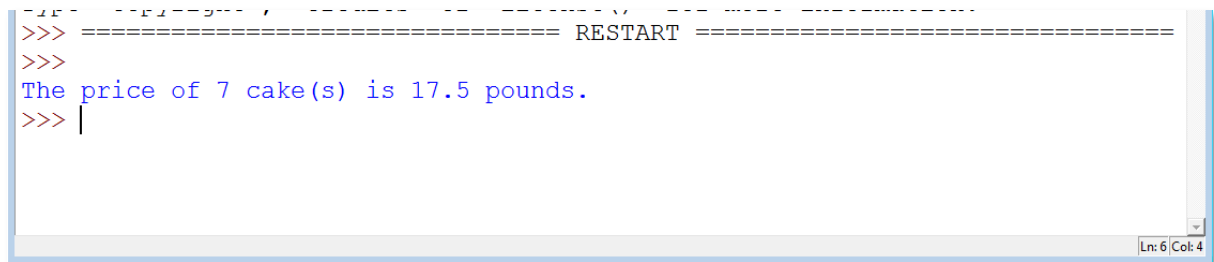


Figure 15

What if I want to compute the bill for nine cakes? In that case I just have to change the value 7 to 9 in the module, save the changes, and run the program again. Figure 16 shows the new result.

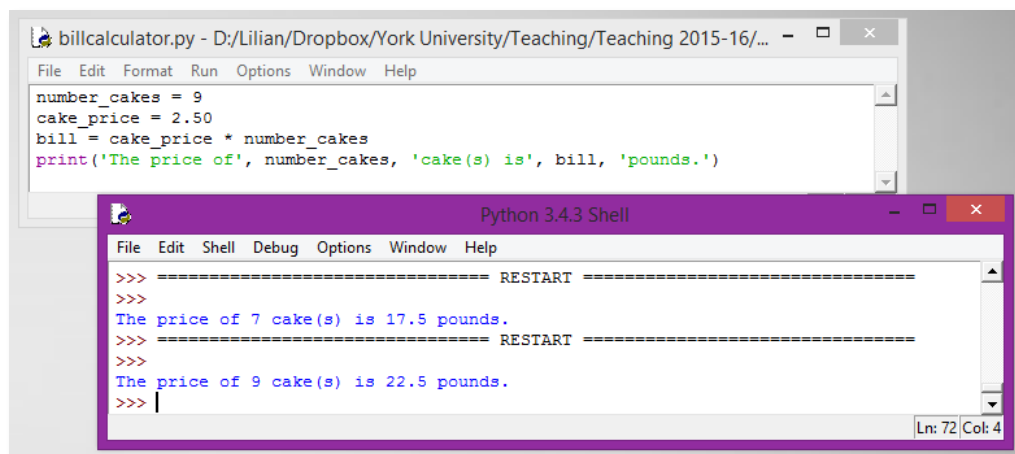


Figure 16

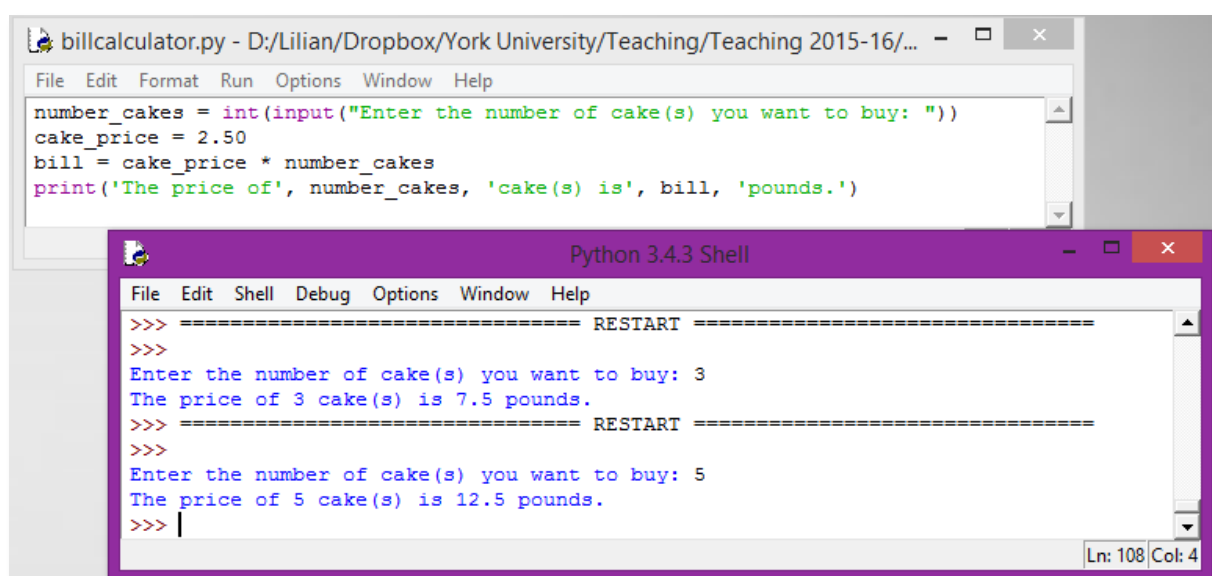
User input

Using modules is an improvement, however every time I want to change the number of cakes, I need to change a value in my code, save the changes and run the program. This is not satisfactory. It would be much better if I could use a statement to ask users to enter new values without changing the code.

Luckily such a statement exists, it uses the function `input(string)`. If you are looking at the first line of code in Figure 17, the statement uses the function `input`. How does it work?

- `input` is the name of the function we want to use (same as for `len`)
- Between parentheses we have a string containing the message we want to display to the user, it is called a **parameter** of the **function** `input`.
- On the left hand side of the assignment is a variable named `number_cakes`,
- It means that the **value returned** by the function `input` will be **assigned/stored** into the variable `number_cakes`.
- The value entered by the user can be used via the variable `number_cakes`.
- The returned value of the `input` function is always string (even if you enter 3.0) via the keyboard. For this reason we convert (cast) the returned value into an integer using the cast operator `int(...)`.

Figure 17 shows the values returned by the program when run twice (without changing the code), and where the user entered two different inputs (3 and 5). This is an important step, we now have a single program that can return different values depending on user input (user being human or machine) without any change in the code.



The screenshot shows a Python IDE with two windows. The top window, titled 'billcalculator.py', contains the following code:

```
number_cakes = int(input("Enter the number of cake(s) you want to buy: "))
cake_price = 2.50
bill = cake_price * number_cakes
print('The price of', number_cakes, 'cake(s) is', bill, 'pounds.')
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the script. It displays the prompt 'Enter the number of cake(s) you want to buy: 3' followed by the output 'The price of 3 cake(s) is 7.5 pounds.' and then 'Enter the number of cake(s) you want to buy: 5' followed by the output 'The price of 5 cake(s) is 12.5 pounds.'

Figure 17

Control Structures 1

The programs that we can achieve so far are not very interesting. Every command is executed exactly once. This is very limited. We are now going to introduce **control structures** so statements might not be executed, executed once or many times.

The first control structure we will look at will enable us to choose which statement to execute depending on a condition. Let look at our latest program. For our output we have to print cake (s) in case we have one or more cakes. It will be better if we could decide to print cake or cakes depending on the number of cakes entered by the user.

In plain English, we could write:

IF *the number of cake is one*, print cake, OTHERWISE/ELSE print cakes

You can notice that the underline part of the sentence is a condition, e.g. is True or False, and depending on the value we will be doing two different things. If it is **True**, we will print cake (singular), if it is **False** we will print cakes (plural). So depending on the value of the condition, one statement will be ignored and the other will be executed, it is sometime called branching.

How is it done in Python?

We have the if-else control structure.

```
if condition :  
    Statement A1  
    ...  
    Statement An  
else :  
    Statement B1  
    ...  
    Statement Bk
```

First thing to remember are the two keywords, `if` and `else` with the colon at the end of the line. The second is the condition, an expression that can be evaluated to **True** or **False**. Such expressions are called **Boolean expression** (can return only the value `True` or `False`). Finally a series of statements after the `if` and the `else`. Note, and this is important, that the statements are **indented to the right** compared to the `if` and the `else`. Indentation is Python's way of representing blocks of code. The expression above means, if condition is `True` then execute block of code `A1` to `An`, else execute block of code `B1` to `Bk`.

Example

Figure 18 shows of an if-else statement can be used to improve our program. The condition is `number_cakes == 1`. Note the use of **double equal operator**; it is to inform the interpreter that it is not an assignment but an equality comparison (you will make this error often). In our case the blocks of statements contains only one statement each, but they could have had more than one.

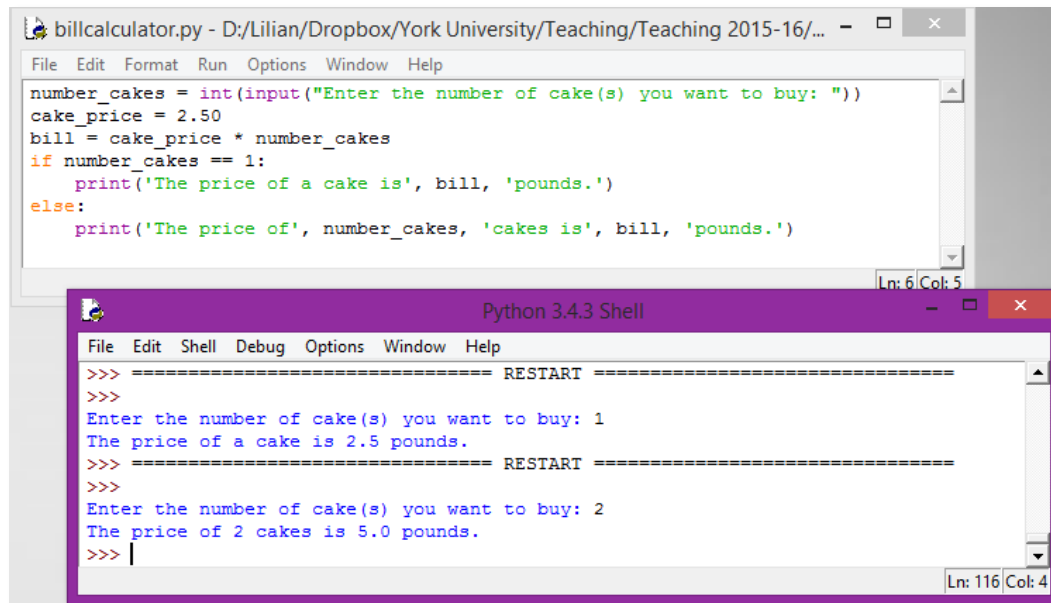


Figure 18

We can see in Figure 18 the result of running the program twice using two different inputs. The results obtained are the expected ones. The introduction of control structures, and especially branching, presents a new problem: TESTING!

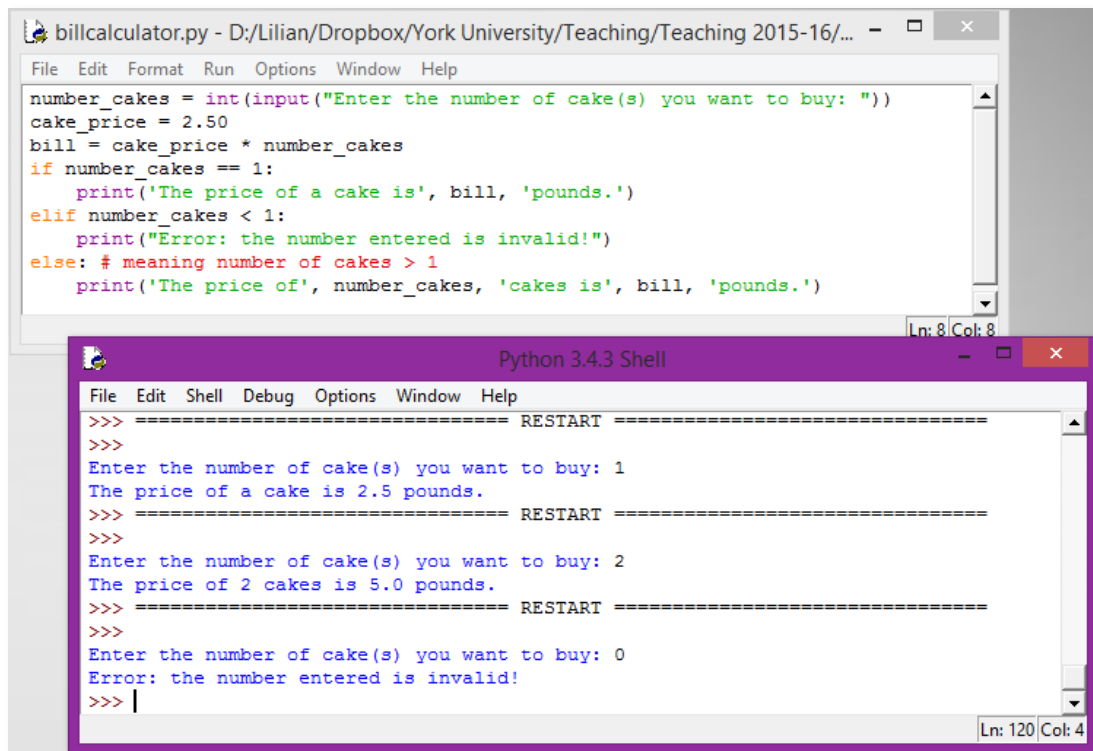
To ensure that our program works properly, we must design a series of tests such that all branches of our program are traversed. We will look at testing in more details later in the term; nonetheless you should keep that problem in mind for the time being.

Be wary of the mighty user

When you are interacting with a user, you can be sure it will input something you did not expect. How can I protect my program? In our example we assumed that the user will input a value greater or equal to 1. What if the input is -1? The program will execute without errors/crashing, however the result will be incorrect/not making sense. We can use another control structure: `if-elif-else`.

```
if conditionA :
    Statement A1
    ...
    Statement An
elif conditionB :
    Statement B1
    ...
    Statement Bm
else :
    Statement C1
    ...
    Statement Ck
```

An example on how to use the `if-elif-else` control structure for our problem is shown in Figure 19. You can use as many `elif` between the `if` and `else` as you need (almost). Now we can consider that our program is protected (to a certain extent) against user inputs. If the user enters an incorrect number we are displaying an error message, otherwise we provide a correct answer. Note again that our test case is made of three inputs in order to go through all branches of our program, e.g. the `if` branch, the `elif` branch, and the `else` branch.



The image shows two windows from a Python IDE. The top window, titled 'billcalculator.py', contains the following Python code:

```
number_cakes = int(input("Enter the number of cake(s) you want to buy: "))
cake_price = 2.50
bill = cake_price * number_cakes
if number_cakes == 1:
    print('The price of a cake is', bill, 'pounds.')
elif number_cakes < 1:
    print("Error: the number entered is invalid!")
else: # meaning number of cakes > 1
    print('The price of', number_cakes, 'cakes is', bill, 'pounds.')
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the program. It displays three test cases separated by 'RESTART' markers:

```
>>> ===== RESTART =====
>>> Enter the number of cake(s) you want to buy: 1
The price of a cake is 2.5 pounds.
>>> ===== RESTART =====
>>> Enter the number of cake(s) you want to buy: 2
The price of 2 cakes is 5.0 pounds.
>>> ===== RESTART =====
>>> Enter the number of cake(s) you want to buy: 0
Error: the number entered is invalid!
>>> |
```

Figure 19

NOTE: what is the statement in red after the `else`? In Python, as in all languages, we can have comments. A comment is just plain text that will be ignored by the interpreter, e.g. not executed. They are mainly there to help understanding the code, especially if you are reading someone else's code. The symbol `#` is used in Python (it differs between languages, e.g. Java, C++, etc.). Everything that follows `#` and is on the same line of code is considered a comment and will be ignored by the interpreter. In **Idle** comments are highlighted in **red**.

- Search how block of comments are represented in Python.