



University of
South Australia

Problem Solving and Programming

Week 5 – Functions

- Debugging

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of South Australia** in accordance with section 113P of the *Copyright Act 1968* (**Act**).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Python Books

Course Textbook

Gaddis, Tony. 2015, *Starting Out with Python*, 3rd edition, Pearson Education, Inc.

Free Electronic Books

There are a number of good free on-line Python books. I recommend that you look at most and see if there is one that you enjoy reading. I find that some books just put me to sleep, while others I enjoy reading. You may enjoy quite a different style of book to me, so just because I say I like a book does not mean it is the one that is best for you to read.

- The following three books start from scratch - they don't assume you have done any prior programming:
 - The free on-line book "[How to think like a Computer Scientist: Learning with Python](#)", by Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers, provides a good introduction to programming and the Python language. I recommend that you look at this book.
 - There is an on-line book "[A Byte of Python](#)" that is quite reasonable. See the [home page](#) for the book, or you can go directly to the [on-line version for Python 3](#), or [download a PDF copy](#) of the book. This book is used in a number of Python courses at different universities and is another I recommend you look at.
 - Another good on-line book is "[Learning to Program](#)" by Alan Gauld. You can download the whole book in easy to print PDF format, and this is another book that would be good for you to look at.
- If you have done some programming before, you may like to look at the following:
 - [The Python Tutorial](#) - this is part of Python's documentation and is updated with each release of Python. This is not strictly an e-Book, but is book-sized.
 - [Dive into Python 3](#), by Mark Pilgrim is a good book for those with some programming experience. I recommend you have a look at it. You can download a [PDF copy](#).

Problem Solving and Programming

- More on writing programs:
 - User-defined functions
- Debugging your code

Introduction to Functions

User-defined functions

Functions

- Some problems can be very complex.
- In order to solve problems, it may be easier to separate into smaller, simpler tasks.
- This process is called decomposition.
- Functions
 - Represent one idea or perform a single task.
 - We can write our own functions (user-defined functions).

Functions

- A function is a self contained segment of code which implements a specific well defined task.
- Python programs typically combine user-defined functions with library functions available in the standard library.
- A function is invoked by a call which specifies;
 - A function name,
 - May provide parameters/arguments.
- A function may return a value.
- `input` and `print` are examples of standard library functions and are invoked in the following way;

```
print("Welcome to functions!")  
num = input("Please enter number: ")
```

Functions

- How to define a function in Python
 - The general syntax for defining a function is:

```
def functionName(parameters):  
    function_body_suite
```

- Keyword `def` introduces a function definition.
- The `functionName` follows the same rules as variable names.
- The `parameters` are identifier names that will be used as the names *within the function* for the objects passed to the function.
- Parameters are optional, if there is more than one, they are written as a sequence of comma-separated identifiers.
 - Parameters are the information being supplied to a function.
 - Parameters are local to the function.
- The `function_body_suite` is a sequence of Python statements. Must be indented.

Functions

- How to define a function in Python

- The general syntax for defining a function is:

```
def functionName(parameters) :  
    function_body_suite
```

- The `function_body_suite` is a sequence of Python statements. Must be indented.
 - The `function_body_suite` contains:
 - Definitions – variables defined in the function are local variables.
 - Statements.
 - Returning a value from a function is done by a return statement:

```
return expression
```
 - A `return` statement without an expression argument returns `None`.
 - If no return statement is present (i.e. falling off the end of a function), the function also returns `None`.
 - All functions return a value, even if it is the special value `None`.

Functions

- Defining the function does not execute the function – you need to call the function from elsewhere in the program, passing the appropriate information to the function through its parameters.
- Define the functions that you create at the start of your program.
- Only use the information passed to the functions via the parameters or defined within the function.
- Variables that you define inside the function are only visible inside the function, that is, they are local to the function.
 - When you leave the function, you no longer have access to those variables.

Functions

- Example:
 - Write a function which will take two numbers as parameters, sum them, and return the result.

```
def functionName(parameters) :  
    function_body_suite
```

Functions

Solution:

Functions

Example – calling the function once it has been defined:

Functions

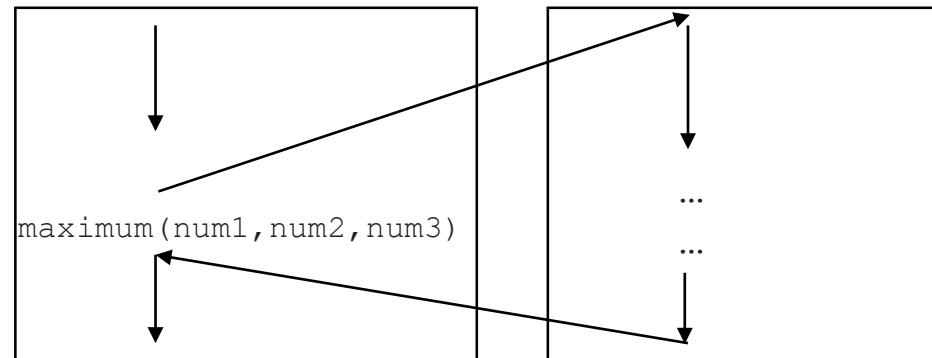
Another Example – find the maximum of three numbers:

```
def maximum( no1, no2, no3 ):  
    maxValue = no1  
  
    if ( no2 > maxValue ):  
        maxValue = no2  
  
    if ( no3 > maxValue ):  
        maxValue = no3  
  
    return maxValue
```

```
num1 = 4  
num2 = 7  
num3 = 1
```

```
result = maximum(num1, num2, num3)  
print('Maximum is: ', result)
```

Order of execution:



Functions

Another Example :

```
def displayInfo():  
    print('This is an example of a function that')  
    print('just prints some text and that\'s about it...')  
  
displayInfo()
```

Output:

```
This is an example of a function that  
just prints some text and that's about it...
```

Functions

- Exercise:
 - Write a function that takes in the final scores of two football teams as parameters and displays either who won the game or whether the game was tied. Refer to the teams as 'Team 1' or 'Team 2'. The function returns nothing.

```
def functionName(parameters):  
    function_body_suite
```


Functions

Solution:

Functions

- Exercise:
 - The built-in `len` function finds the length of a string. Let's write our own function that does the same thing so we can see how the actual `len` might work.
 - Write a function called `length` that takes in a string as a parameter and returns the length of a string. The function returns the length of a string.

```
def functionName(parameters):  
    function_body_suite
```

Functions

Solution:

Functions

Another example :

Write a function called `sumList` that will take in a list of integer numbers as a parameter. The function sums and returns the sum of all the numbers in the list. If the list is empty, the function returns 0.

Functions

Solution:

Functions

Another example :

Create a program which prompts for and reads a mark.

The program calculates the course grade based on the mark entered.

The mark input must be a positive value: 0 – 100 (inclusive).

The program should continue to read marks until the user wishes to quit.

Functions

Solution:

```
# initialize again to 'y'
again = 'y';
```

```
while again == 'y':
```

```
    # prompt for and read mark - initialize loop control
    mark = int(input('Please enter mark: '))
```

```
    # check to confirm that mark is a positive value
    # - test loop control
    while mark < 0 or mark > 100:
```

```
        # display error message to user
        print('Mark must be a positive value 0-100.')
        print(' ')
```

```
        # update loop control
        mark = int(input('Please enter mark: '))
```

```
:
:
```

```
:
:
# determine the students course grade
if mark >= 85:
    result = 'HD'
elif mark >= 75:
    result = 'D'
elif mark >= 65:
    result = 'C'
elif mark >= 55:
    result = 'P1'
elif mark >= 50:
    result = 'P2'
elif mark >= 40:
    result = 'F1'
else:
    result = 'F2'
```

```
# display the result
print('Course grade is: ', result)
print(' ');
```

```
# update loop control
again = input('Enter another mark? (y or n): ')
```

Functions

Solution - using a function to read and validate mark:

Notice that the function does not accept any parameters, but returns a value (mark).

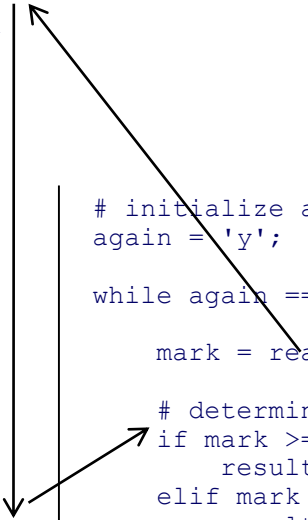
```
def readMark():
    # prompt for and read mark - initialize loop control
    mark = int(input('Please enter mark: '))

    # check to confirm that mark is a positive value
    # - %test loop control
    while mark < 0 or mark > 100:

        # display error message to user
        print('Mark must be a positive value 0-100.')
        print(' ')

        # update loop control
        mark = int(input('Please enter mark: '))

    return mark
```



The diagram consists of a vertical line with two arrows. One arrow points from the `mark = readMark()` line in the right-hand code block up to the `def readMark():` line in the left-hand code block. The other arrow points from the `return mark` line in the left-hand code block down to the `if mark >= 85:` line in the right-hand code block.

```
# initialize again to 'y'
again = 'y';

while again == 'y':

    mark = readMark()

    # determine the students course grade
    if mark >= 85:
        result = 'HD'
    elif mark >= 75:
        result = 'D'
    elif mark >= 65:
        result = 'C'
    elif mark >= 55:
        result = 'P1'
    elif mark >= 50:
        result = 'P2'
    elif mark >= 40:
        result = 'F1'
    else:
        result = 'F2'

    # display the result
    print('Course grade is: ', result)
    print(' ');

    # update loop control
    again = input('Enter another mark? (y or n): ')
```

Debugging your code

Reading

- <http://en.wikipedia.org/wiki/Debugging>

Debugging

- You wrote your program and thought it would work, should work...
- You run it, and find that it doesn't do what you expected it to do...
- What's wrong? How can you fix it?
- That's debugging!
- Debugging is a methodical process of finding and fixing errors in a computer program.
- It is one of the hardest parts of programming.
- Strategy 1: Avoid bugs in the first place by...
 - Careful design (clean decomposition).
 - Errors are minimised through good design.
 - Follow the systematic problem solving strategy.
 - Care with syntactic issues (layout, commenting).
- Strategy 2: Implement in a series of small steps, and test along the way...
 - Bugs are localised to what changed in the program to introduce the bug.
- Finding bugs requires a disciplined, deductive approach.

Debugging

- Typical mistakes made when programming
 - Indentation wrong.
 - Following wrong path (i.e. incorrect boolean expression).
 - Incorrect expression in if statements.
 - Looping the wrong number of times.
 - Looping too few times.
 - Looping too many times.
 - Infinite loop.
 - Incorrect values in variables.
- **All bugs are caused by computers doing exactly what they are told!** The computer will do exactly what you told it to do – you just told it to do the wrong thing! :)
- Attitude makes a big difference when attempting to find and correct a bug.

Characteristics of Good Problem Solvers

- Positive attitude.
 - Belief that academic reasoning problems can be solved through persistence, as opposed to believing either you know it or you don't.
 - Engage a confusing problem.
- Concern for accuracy.
 - Actively work to check your understanding.
- Break the problem into parts.
- Avoid guessing.
 - Don't jump to conclusions.
- Active in problem solving.

Getting Started with a Problem

- “Eighty percent of success is showing up”.
Woody Allen
- “Success is 1% inspiration and 99% perspiration”.
Thomas Edison
- To successfully solve any problem, the most important step is to get actively involved.
 - You must commit to the problem.
 - “Roll up your sleeves”
 - “Get your hands dirty”

Debugging

- My program doesn't work! Can you fix it??
 - Many people seem to get stuck when their program does not work as expected...
 - I don't know what to do... my program just doesn't work!?!
- What **NOT** to do...
 - Make random changes to your code hoping that given the right random change, something will eventually work. This never works!
 - You may introduce new bugs.
 - Try to run the program a few more times in joyful hope that it will start working again!

Debugging

- My program doesn't work! Can you fix it??
 - Many people seem to get stuck when their program does not work as expected...
 - In despair they often seek help with the general statement "My program doesn't work!", followed by a blank, somewhat distressed stare.
- That is like walking into a doctor's surgery and exclaiming "I'm sick, can you make me better?" Or a mechanics and saying "My car doesn't work, can you fix it?"
- Such general statements in no way help the doctor, mechanic or lecturer for that matter know exactly what is wrong or even where to start looking in order to address the problem...
- Think about what could cause the observed behaviour.
- Isolate the bug.
- The bug isn't trying to hide from you... if you are patient and persist, you will eventually find it!
- You need to think like a detective – follow the clues the program gives you to determine the cause of the bug.
- Let's look at a few strategies in order to help you find and fix bugs in your programs...

Debugging

- **Debugging strategies** (in no particular order):
 - Insert debug print statements in your code.
 - Use the IDLE Debugger.
 - Look at what's happening – you will see what the program is doing.
 - Explain the problem out loud to someone else (it doesn't matter if they understand programming or not – it's for your benefit, not theirs).
 - Sleep on it – good for all problem solving!

Debug print statements

- The simplest and most useful debugging tool is the print statement.
- Use it to check variable values at important points in the program.
- For example (should display – Sum is: 6):

```
numbers = [2, 0, 1, 3]
```

```
count = 0
```

```
for no in numbers:
```

```
    count = no
```

```
print("Sum is:", count)
```

...the above program displays – Sum is: 3

- Print the pre-assignment values and post-assignment result:

```
print("count", count, " + number: ", no)
```

```
count = no
```

```
print("count after: ", count)
```

- Catches errors such as:

```
count = no
```

Debug print statements

- Use it to check variable values at important points in the program.
- For example (should display – Sum is: 6):

```
numbers = [2, 0, 1, 3]
count = 0

for no in numbers:
    print("count", count, " + number: ", no)
    count = no
    print("count after: ", count)

print("Sum is:", count)
```

- Catches errors such as:

```
count = no
```

- Should be:

```
count = count + no
```

- Remove debug statements once finished.

Debug print statements

- Use it to determine how far the execution of your program got before producing the error or terminating abnormally by including descriptive "I am here" statements.
- Use it to determine flow of control errors – e.g. incorrect boolean condition, etc.
- For example (should display – Count is: 5):

```
max = 5
count = 0
k = 0

while k > max:
    count = count + 1
    k = k + 1

print('Count is: ', count)
```

...the above program displays – Count is: 0

Debug print statements

- Print descriptive "I am here" statements to ascertain what is actually going on in your program.

```
max = 5  
count = 0  
k = 0
```

```
print("before while loop")
```

```
while k > max:
```

```
    count = count + 1
```

```
    k = k + 1
```

```
    print("in while loop")
```

```
print('Count is: ', count)
```

Output:

before while loop

Count is: 0

Notice in output that
in while loop
is not printed.

- Catches errors such as:

```
while k > max:
```

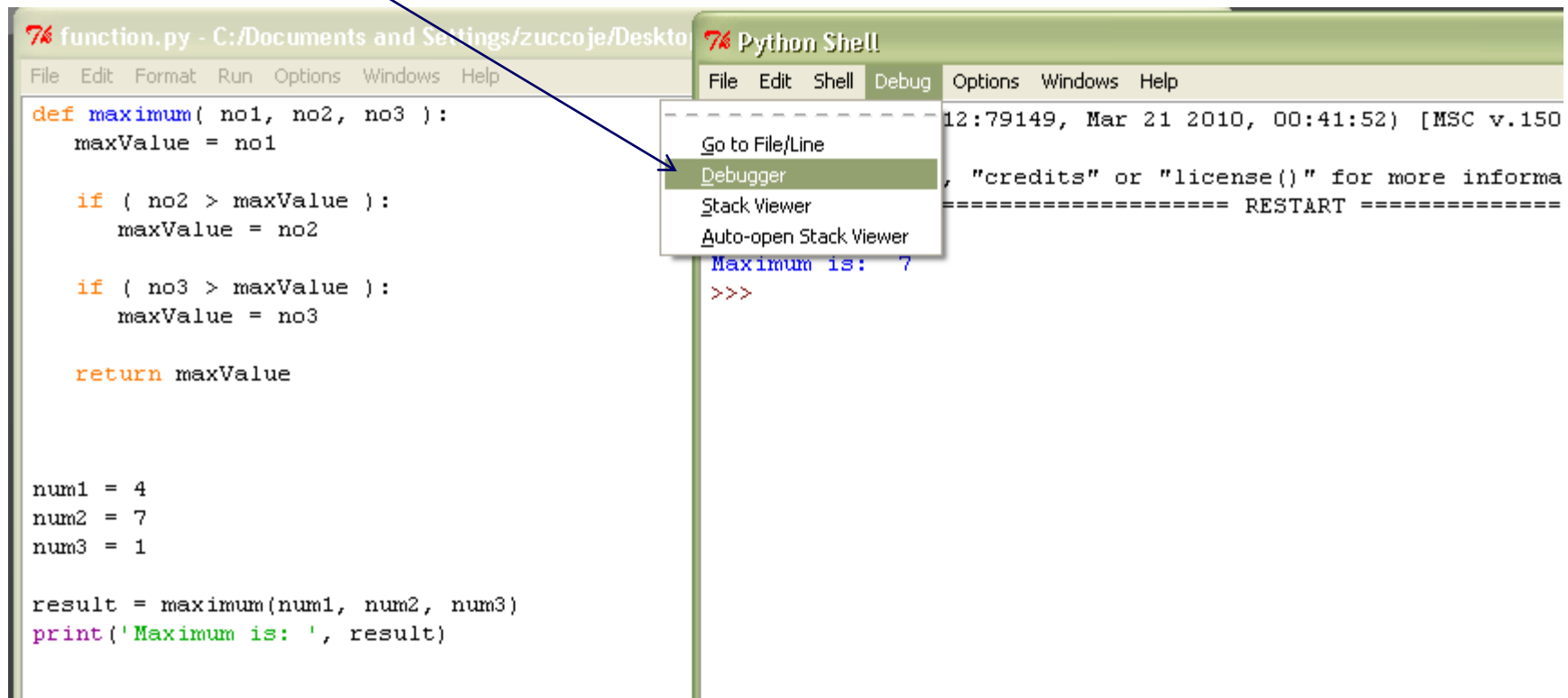
- Should be:

```
while k < max:
```

- Remove debug statements once finished.

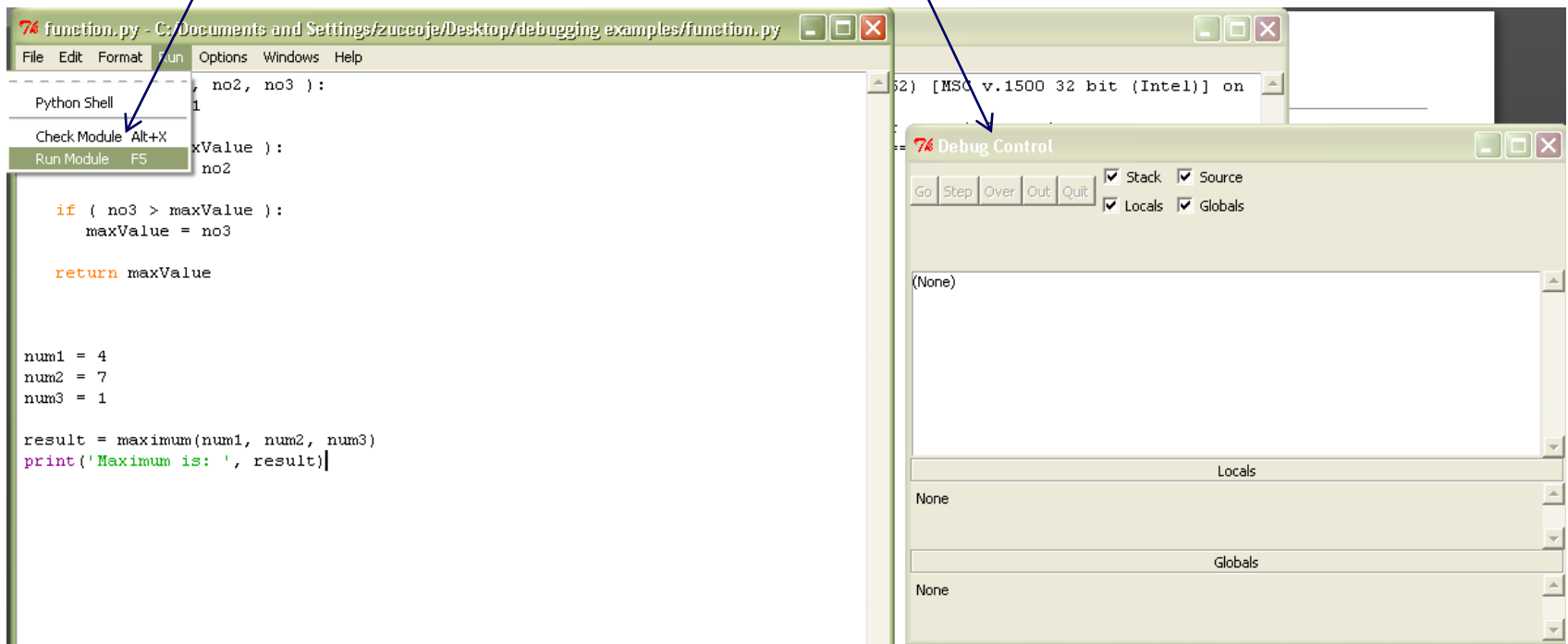
Use the IDLE Debugger

- Another approach to finding and fixing errors in your code is to use the debugger which is part of IDLE.
- To start debugging
- Click Debugger on the Debug menu.



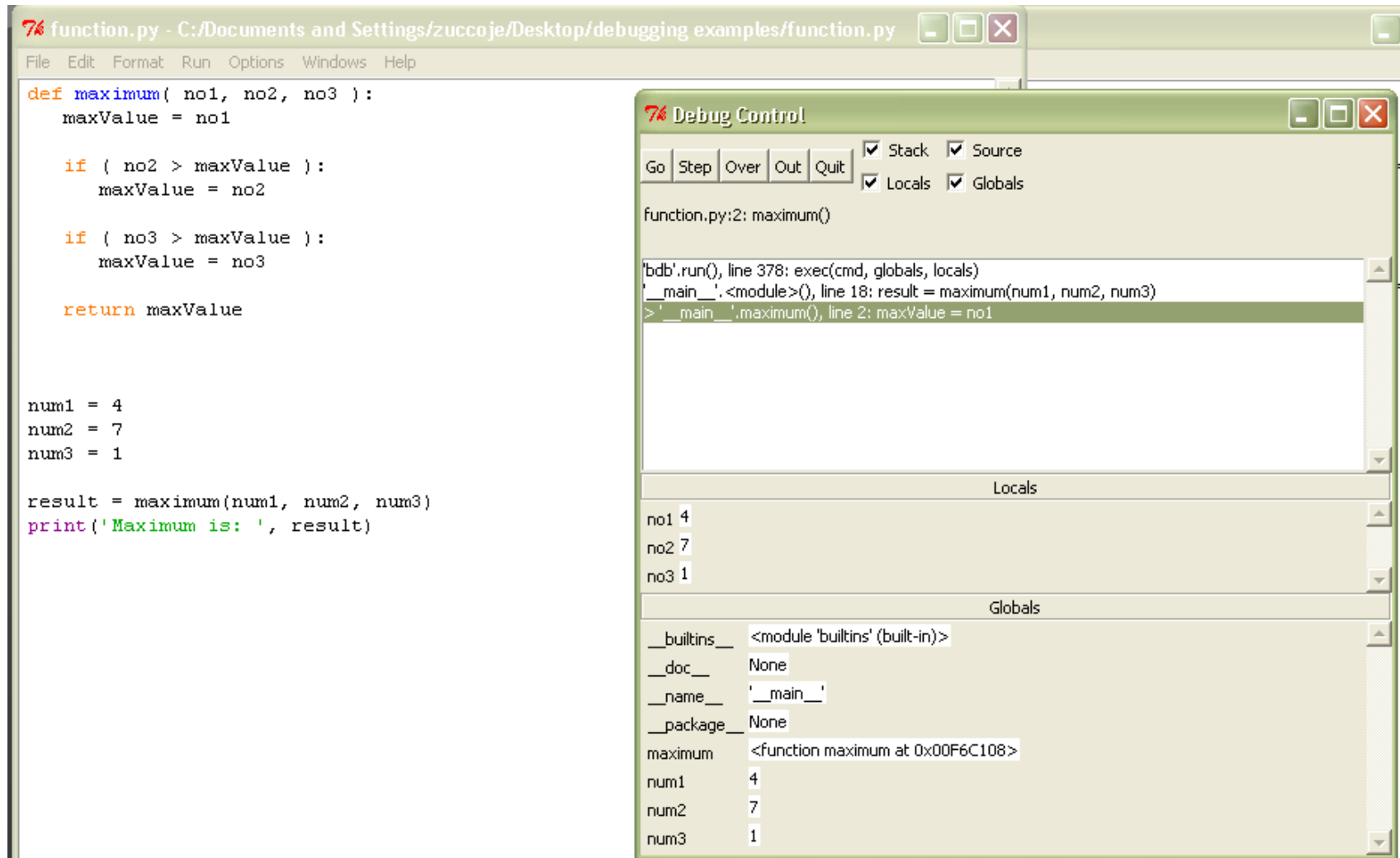
Use the IDLE Debugger

Doing so will display the following Debug Control window.
Click Run Module on the Run menu.



Use the IDLE Debugger

- You can now see information about your code in the Debug window.



The screenshot displays the IDLE Python IDE interface. The main window shows a Python script named `function.py` located at `C:/Documents and Settings/zucchoje/Desktop/debugging examples/function.py`. The script defines a `maximum` function and calls it with three arguments.

```
def maximum( no1, no2, no3 ):
    maxValue = no1

    if ( no2 > maxValue ):
        maxValue = no2

    if ( no3 > maxValue ):
        maxValue = no3

    return maxValue

num1 = 4
num2 = 7
num3 = 1

result = maximum(num1, num2, num3)
print('Maximum is: ', result)
```

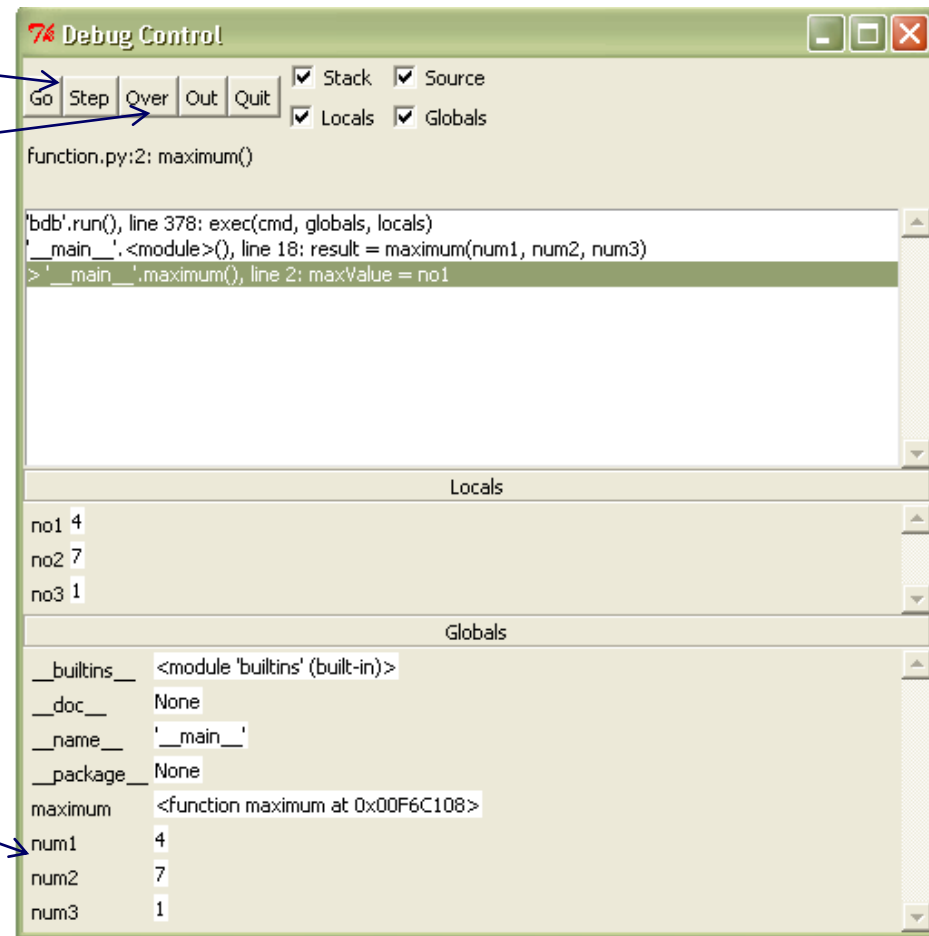
The **Debug Control** window is open, showing the current execution state. It includes buttons for `Go`, `Step`, `Over`, `Out`, and `Quit`, along with checkboxes for `Stack`, `Source`, `Locals`, and `Globals`. The `function.py:2: maximum()` is selected in the call stack. The `Locals` pane shows the current values of `no1` (4), `no2` (7), and `no3` (1). The `Globals` pane shows the state of global variables, including `__builtins__`, `__doc__`, `__name__`, `__package__`, `maximum`, `num1`, `num2`, and `num3`.

Locals	
no1	4
no2	7
no3	1

Globals	
__builtins__	<module 'builtins' (built-in)>
__doc__	None
__name__	'__main__'
__package__	None
maximum	<function maximum at 0x00F6C108>
num1	4
num2	7
num3	1

Use the IDLE Debugger

- Brief explanation of the menu options:
 - Step
 - Single-steps through instructions in the program, and enters each function call that is encountered.
 - Over
 - Single-steps through instructions in the program. If this command is used when you reach a function call, the function is executed without stepping through the function instructions.
- You can watch the contents of any variables as your program executes. This is useful because if the value changes in an unexpected way, you'll see it as it happens.
- A good strategy is to watch variables over several lines of code to observe when they change.



Explain your code

- Explain the problem out loud to someone else (it doesn't matter if they understand programming or not – it's for your benefit, not theirs).
- Stating your problems out loud, sometimes makes the solution more clear.
- Explain your code, line-by-line to someone else... or a rubber duck (http://en.wikipedia.org/wiki/Rubber_duck_debugging) – it doesn't really matter.
- The theory is that if you explain the code out loud, it is expected that when you come across a piece of code that is incorrect you will realise it.

Sleep on it

- Passage of time can un-stick many problems.
- The mind “incubates” the problem.
 - Perhaps works on the problem unconsciously.
- Each of us has circumstances in which we are most creative.
 - Driving to uni, waiting for an appointment, taking a shower.
 - Take advantage of this.
- Must give your self time to solve the problem.
- It gives you a chance to come at the problem with another approach.
 - Does the solution occur to you?
 - Perhaps a new approach that immediately leads to the solution?
- But you must start programming with enough time to allow yourself to sleep on it.

Sleep on it

- Get enough sleep...
 - It is very difficult to think clearly when you are sleep deprived.
- Work away from the computer screen.
- Think about your program and what it should do.
- Some times it helps to just walk away and do something different - even for a few hours.

Debugging

- General rules:
 - Debug section by section. If you have finished coding all the program, comment out most it and start checking from the very beginning.
 - Always know the expected output of the section you are debugging before you start.
 - A “print” statement is a good tool for debugging. Print all necessary messages so that you know what your program is doing. Often a program behaves differently when it first runs to what the programmer expects.
 - Think deeply, methodically and logically. Keep track of what you have already tested.
 - Test one idea at a time. If you change several things and it now works, which one made the difference?
 - Challenge your own presumptions. Parts that you ‘know’ are right may not be.
 - Debugging is like building a car. You need to test the components before you test the whole car.

End of Week 5