# Introduction to Programming

## Lab Worksheet

### Week 9

Prior to attempting this lab tutorial ensure you have read the related lecture notes and/or viewed the lecture videos on MyBeckett. Once you have completed this lab you can attempt the associated exercises.

You can download this file in Word format if you want to make notes in it.

You can complete this work using the Python interpreter in interactive mode. This could be inside an IDE, or just a command prompt.

**Topics covered:**

* Exception Handling
* Types of Exceptions
* Using ‘else’ and ‘finally’
* Raising Exceptions

For more information about the module delivery, assessment and feedback please refer to the module within the MyBeckett portal.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## ©2021 Mark Dixon

## Modified 2021 Tony Jenkins

## 

## **Exception Handling**

Although many errors can be avoided by changing the program code, there are some circumstances that may result in error conditions, which are outside the control of the programmer. Most modern languages provide a mechanism known as *exception handling*. This is an approach to dealing with certain types of errors that can occur at *run-time*.

Problems that occur during execution are often reported as *exceptions*, and although they can not always be avoided, code can be written that detects and *handles* exceptions. If a program does not *handle* an exception then it will result in program termination and an error being generated in the form of a *Traceback* message. Such messages are designed to allow programmers to fix issues, and should never really be seen by the end-user. Hence it is the programmer’s job to ensure they *handle* the *exceptions* that can occur and report them in a graceful way.

The exception handling mechanism is supported in Python using a try block. If any code placed within a try block *raises* an exception, it can be handled within an associated except block.

For example, the following try code block contains a call to the int() function, which converts a string into an integer type value. If the given string argument does not represent a valid integer, then an *exception* is *raised* causing the code within the except block to execute.

try:

value = input("Enter a number: ")

num = int(value)

print("That number squared is", num \* num)

except ValueError:

print("The value entered is not a number")

By dealing with the exception, a sensible message can be shown and program execution can continue, rather than immediate program termination followed by a traceback message. If an exception does not occur within the try block, then all of the remaining statements are executed as normal and the except block is skipped.

The program jumps to the except block as soon as the exception is raised, so any statements remaining in the try block are never executed. As will be shown later, it is possible to have multiple except blocks to handle different exceptions.

**TASK**: Use an editor to write a program that asks the user to input two numbers, converts them to floats, then displays their product. Include a try...except block to ensure any invalid numbers are dealt with gracefully. Execute the program using the Python interpreter for testing.

**Types of exceptions**

The except statement is accompanied by an *exception name*. This refers to the *type* of exception we want to handle within the code block. In the previous example the type of exception handled was a ValueError, which indicates an invalid value has been detected.

Some of the most commonly *raised* exception names include -

* ZeroDivisionError

Raised when the second argument of a division (/) or modulo operation (%) is zero, e.g. 10/0

* FileNotFoundError

Raised when a file or directory is requested but doesn’t exist.

* ValueError

Raised when an operation or function receives an argument that has the right type but an inappropriate value, e.g. int("xyz")

* IndexError

Raised when a sequence index value is out of range, e.g. names[len(names)]

* KeyboardInterrupt

Raised when the user hits the interrupt key (normally Control-C)

Having multiple exception types allows us to explicitly identify different types of errors that have occurred, and if required handle them independently.

In order to allow exceptions to be handled separately it is necessary to associate multiple except statements with a single try statement. For example, the following code handles two different types of exception.

try:

dividend = input("Enter the first number: ")

divisor = input("Enter the second number: ")

print("First divided by second is", float(dividend)/float(divisor))

except ZeroDivisionError:

print("The second value was 0 (cannot divide by 0)")

except ValueError:

print("Entered values must be numbers")

To see the various built-in exceptions visit <https://docs.python.org/3/library/exceptions.html>

In practice, to find what exception will be raised when some error condition is detected it is usually easiest to provoke the error and examine the output!

**Adding an else block**

In some circumstances it is useful to be able to execute code following a try block when an exception was not *raised*. This is possible by extending a try...except block with an associated else statement. Code within the else block is only executed if an exception *does not occur*.

try:

dividend = input("Enter the first number: ")

divisor = input("Enter the second number: ")

quotient = float(dividend)/float(divisor)

except ZeroDivisionError:

print("The second value was 0 (cannot divide by 0)")

except ValueError:

print("Entered values must be numbers")

else:

print("First divided by second is", quotient)

Within this example, the final print() function is only called if neither of the exceptions occurred.

**TASK**: Use an editor or your IDE to write a program that creates the List of names shown below.

names = ["Terry", "John", "Michael", "Eric", "Terry", "Graham"]

Use the input() function to prompt the user to enter a number. Convert that number to an integer then use it to access one of the names within the list. Assign this to a variable called name, like so:

name = names[int(num)]

Use a try...except block to handle and report the exceptions that may be *raised* (hint: there are two possible exception types). Include an else block that prints the name that was selected, but only when an exception did not occur.

Execute the program using the Python interpreter for testing.

**Clean-up statements**

When dealing with exceptions is it sometimes necessary to perform ‘clean-up’ actions. These are statements that must always execute, whether an exception occurred or not. This can be achieved using a finally block, which must appear as the last part of a try...except...else block. Unlike the else block, the code within a finally block always executes. These statements will execute even if the try block contains a return statement.

Such actions are usually used to release important resources, a very typical example is the closing of a file, since this must always be done.

**TASK**: Use a text editor or your IDE to enter the code below and save it to a file called divider.py. Execute the program using the Python interpreter. When prompted, try inputting various values, some of which cause exceptions to be raised, and some that don’t. Notice the output shown.

try:

dividend = input("Enter the first number: ")

divisor = input("Enter the second number: ")

quotient = float(dividend)/float(divisor)

except ZeroDivisionError:

print("The second value was 0 (cannot divide by 0)")

except ValueError:

print("Entered values must be numbers")

else:

print("First divided by second is", quotient)

finally:

print("The divider program has finished")

When developing a program the decision has to be made about which exceptions should be handled. In many circumstances *traceback* messages are shown to report issues because the program code is actually incorrect, this is especially true for SyntaxError type exceptions.

It is therefore important to be able to distinguish between exceptions that should be handled within a try...except block, and those that can be avoided altogether by changing the program code. If the syntax and logic of a program is correct and the exception could still potentially occur during execution, then a try...except block should usually be used.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## **Raising Exceptions**

All reasonable size programs should include code to handle exceptions. The exception mechanism is a very elegant and consistent way of dealing with “error logic”. As well as just handling exceptions, it is also possible to write code that causes the exceptions to occur in the first place. i.e. your own code can *raise* exceptions.

When writing code that contains many functions, or writing modules containing functions designed to be reused, the *raising* of exceptions is the best way to report problems to the caller. As we have seen, the built-in functions and the modules within the Python Standard Library use the exceptions mechanism extensively. Hence, it is good practice to follow the same style of programming when developing your own programs.

*Note: it may seem obvious that error messages should be printed to the screen, but remember there will not always be a screen, or indeed anyone to read an error message. Exceptions can be converted into error messages for users, or processed by other programs in some way.*

An exception can be *raised* in Python by using the raise keyword, followed by the name of the exception type.

num = int(input("Enter a number between 1 and 10: "))

if num < 1 or num > 10:

raise ValueError

It is also possible to pass parameters when an exception is *raised*, this can often be used to provide more information about the problem:

num = int(input("Enter a number between 1 and 10: "))

if num < 1 or num > 10:

raise ValueError("The number must be between 1 and 10")

The information that is passed during the *raising* of an exception can then be accessed by the try...except block that *handles* the exception.

try:

# call some function that can raise a ValueError exception

except ValueError as err:

print("The following problem occurred:", err)

Notice the additional as err statement , this provides access to the exception object, which in this case contains the message to be printed.

When raising exceptions it is usual to choose the built-in exception that is closest to the cause of the error. (It is possible to define further exception types, but that is beyond our scope here.)

**TASK**: Use your IDE or a text editor to write a small program that defines a function called get\_name(). This function should prompt the user to enter their name, which it should then return. However, the function should first check whether the entered name was at least 4 characters in length. If not, it should raise a ValueError exception.

Once the function is finished, add additional code to the program that makes a call to the function. Place the call within a try...except block that *handles* and prints details about the exception. If an exception does not occur then the entered name should be printed to the screen. e.g. "Your name is "….

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## 

## **Key Terminology**

Understanding key terminology is important not only when programming but for understanding computer science in general. Certain terms, technologies and phrases will often re-occur when reading or learning about many computer science topics. Having an awareness of these is important, as it makes learning about the various subjects and communication with others in the field easier.

**TASK**: Look at each of the phrases below and ensure you understand what each of these means. For any that you do not understand, do a little research to find a definition of each term. This research may involve looking back over these notes, or the associated lecture notes. It may also involve searching for these terms on the Internet.

* Exception Handling
* Traceback
* Exception Type
* Raising an exception
* Finally clause

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## **Practical Exercises**

You have now completed this tutorial. Now attempt to complete the associated exercises.