> *My hovercraft is full of eels*

Error Handling

We have seen many errors that the interpreter has produced when we have done something wrong. In this case, we say that the interpreter has raised an error. A typical example is when we wish to use user input. We cannot guarantee that the user will input values correctly. If we are not careful, user input might cause an error. When this happens, we want to be able to catch the error (Exception) and deal with it ourselves (users do not like to see red).

To be able to do this we surround the code that is likely to raise an error in a try, except statement. The following gives a couple of examples of **error handling** on the int function that attempts to convert user input into an integer. To do this we shall write a small function definition as follows.

```
def int_exception() :
    """Asks for user input and attempts to convert it into an integer.

    Returns:
        int: Integer value entered by user;
             or -1 if they did not enter an integer.
    """
    num = input("Enter a number: ")
    try :
        return int(num)
    except Exception :
        print("{0} is not a number".format(num))
        return -1
```

This function simply asks the user for a number. It then attempts to convert the input into an int and return it. See how the return int(num) is inside a try statement and followed by an except Exception statement. This is how the exception is caught. If the block of code between the try and except lines raises an exception then the rest of the block is not executed and the code after the except line is executed instead. In this case, we print a short message and return -1.

Saving the file as int_exception.py we can perform a couple of test as below.

```
>>> int('10')
10
>>> int_exception()
Enter a number: 10
10
>>> int('ten')

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    int('ten')
ValueError: invalid literal for int() with base 10: 'ten'
>>> int_exception()
Enter a number: ten
'ten is not a number'
-1
```

The first example simply shows that the string '10' can be turned into an integer. The next example shows our function and we give 10 as user input and see that it returns 10. The next example shows then the string 'ten' raises an error as it cannot be converted into a string. We then run our function again and see that we do not get the error but instead get the message 'ten is not a number' and -1 is returned.

Note how we used except Exception, this catches ***any*** error that happens. It is OK to do this in this case as there are not many other ways this code can raise an error. It is **not always good practice to do this**, especially in complex code. Sometimes we end up catching errors raised by mistakes in our code, not the user input. It would therefore, be better to state the exception we want to catch rather than the "catch all" Exception.

Exception Handling Syntax

```
try :
    body1
except ExceptionName :
    body2
```

Semantics

Exception handling starts with a `try` statement followed by a colon. Then on a new line, and indented, is *`body1`*, this is the body of code that is to be tested for exceptions. This is followed, on a new, de-dented line, by an `except` statement followed by the *`ExceptionName`* that is to be caught, which is followed by a colon. On a new line, and indented, is *`body2`*, this is the body of code to be run when the exception is caught. If all errors are to be caught then the *`ExceptionName`* to use is `Exception`.

Using the Exception

It is possible to store the exception caught so that information is able to be extracted for use. This is done using as. To demonstrate this we are going to modify our code from above to look like the function definition below.

```
def int_exception(in_num) :
    """Asks user for a number and divides 'in_num' by the input.

    Parameters:
        in_num (int): Number to be divided by user input.

    Returns:
        float: 'in_num' divided by the number entered by the user;
               or -1 if input is not an integer or if input is 0.
    """
    num = input("Enter a number: ")
    try :
        num = int(num)
        return in_num / num
    except Exception as e :
        print("Error: {0}".format(str(e)))
        return -1
```

The function now takes a number argument. It still asks the user for a number. The number from the argument is then divided by the number given by the user and returned. If an error is

raised then it is caught and assigned to `e`. An error message is then returned including the string representation of the error.

Saving the file as int_exception2.py we can perform a couple of test as below.

```
>>> int_exception(15)
Enter a number: 10
1.5
>>> int_exception(15)
Enter a number: ten
Error: Invalid literal for int() with base 10: 'ten'
-1
>>> int_exception(15)
Enter a number: 0
Error: division by zero
-1
```

The first example works as expected, the function returns 1.5. The next two examples show the function outputting an error message; the first being 'Error: could not convert string to float: ten' from a ValueError, which is from attempting to float a non numerical string from the input and the other is 'Error: float division by zero' from a ZeroDivisionError, which happens if 0 is input (we can not divide by zero).

Exception Handling Syntax

```
try :
    body1
except ExceptionName as var :
    body2
```

Semantics

This works the same as using just try and except. The difference is that the exception, *ExceptionName*, is assigned to the variable *var*.

Catching Different Exceptions

We have seen that it is possible to catch every error raised and that we can get information about the error. So how do we catch a specific error that we know is possible from incorrect input? It is possible to specify what errors we want to catch and deal with. This way we only catch the errors that are caused by incorrect input, rather than everything and miss the errors raised by any incorrect code.

To demonstrate this let's look at the function definition below:

```
def get_numbers(dividend) :
    """Asks the user repeatedly for numbers and calculates 'dividend'
    divided by each number.

    Results of division are stored in a list.
    The list is returned if nothing is entered.

    Parameters:
        dividend (int): Number to be divided by the numbers entered by user.

    Return:
        list<float>: List of dividend divided by each input number.
    """
    results = []
    while True :
        num = input("Enter Number: ")
        if not num :
            break
        try :
            num = float(num)
            results.append(dividend / num)
        except ValueError :
            print("That is not a number")
        except ZeroDivisionError :
            print("Can't divide by zero")
    return results
```

This function takes a number as an argument. It then repeatedly asks for user input with input. If there is nothing typed then the while loop breaks. If something is typed then it attempts to convert the input to a float and then attempts to divide the argument number by the input number, appending it to the list. It checks for two different errors, ValueError and ZeroDivisionError In both cases a simple message is printed to tell the user that they have given wrong input.

After saving the file as get_numbers.py we can run a few tests.

```
>>> get_numbers(10)
Enter Number: 2
Enter Number: a
That is not a number
Enter Number: 4
Enter Number: 0
Can't divide by zero
Enter Number: 3.5
Enter Number:
[5.0, 2.5, 2.857142857142857]
```

It can be seen in this example that if non numerical inputs are given then we get That is not a number. If 0 is input then the message is Can't divide by zero. Any numerical based inputs in this test work.

Exception Handling Syntax

```
try :
    body1
except ExceptionName1 :
    body2
.
.
.
except ExceptionNameN :
    bodyn
```

Semantics

This starts the same as exception handling only there are repeated except statements for every error that is to be caught.

Multiple `except` statements can have a combination of assigning the exception to a variable or not.

Dealing with Unknown Exceptions

Sometimes we need to be prepared for any exception that may happen. We have seen a way of dealing with this in the first couple of exception handling examples with except Exception. However, sometimes there are errors that we wish to deal with specially and just have a single case for any other exception that may occur. We can modify the function definition above to be able to demonstrate this.

```
def get_numbers(dividend) :
    ""Asks the user repeatedly for numbers and calculates 'dividend'
    divided by each number.

    Results of division are stored in a list.
    The list is returned if nothing is entered.

    Parameters:
        dividend (int): Number to be divided by the numbers entered by user.

    Return:
        list<float>: List of dividend divided by each input number.
    """
    results = []
    while True :
        num = input("Enter Number: ")
        if not num :
            break
        try :
            num = float(num)
            results.append(dividend / num)
        except ValueError :
            print("That is not a number")
        except ZeroDivisionError :
            print("Can't divide by zero")
        except Exception as e :
            print("Unknown Error {0}".format(str(e)))
            return []
    return results
```

This function definition is the same except that we have added an extra except statement. This except statement is another except Exception like we have seen before. The way this function now works is, if a ValueError or ZeroDivisionError error occurs then it will behave the same as the previous example. If any other exception occurs then the last except statement will catch it and print out an error message and return an empty list.

Saving now as get_numbers2.py we can test the function.

```
>>> get_numbers(10)
Enter Number: 2
Enter Number: f
That is not a number
Enter Number: 0
Can't divide by zero
Enter Number:
[5.0]
>>> get_numbers("g")
Enter Number: 2
Unknown Error: unsupported operand type(s) for /: 'str' and 'float'
[]
```

The first few examples are as before. The last example is an example of the new functionality. The function itself was given a string argument when it was meant to be given numbers only. This created a different exception when we attempted to divide a string by a number. Therefore, our function prints an error message and returns an empty list.

Raising Exceptions

There are many situations where we might want an error to occur. If a function receives incorrect input, or some other invalid action occurs, it is better to let the function raise an exception, which forces another part of the code deal with the problem (by using `try-except` statements).

As an example, we will revisit the prime numbers example from earlier. To test if a number is prime, we wrote this function:

```
def is_prime(num) :
    """Returns True iff 'num' is prime.

    Parameters:
        num (int): Integer value to be tested to see if it is prime.

    Return:
        bool: True if 'num' is prime. False otherwise.

    Preconditions:
        num > 1
    """
    i = 2
    while i < num :
        if num % i == 0 :
            return False
        i = i + 1
    return True
```

We discussed that the precondition `n > 1` means that inputs which don't fit this criteria could have unknown consequences. This function would return `True` if the input was invalid, and it

would be the user's responsibility to check that the input was valid. If the user doesn't do this, there could be more disastrous consequences later in the program. We will modify the function to **raise** an error when the input is invalid; this will make sure any mistakes don't quietly pass by.

Different types of exception in Python serve different purposes. For example, we use `ValueError` to represent an inappropriate value (for example, if `n <= 1`), and `TypeError` to represent an input of the wrong type (for example, a `float`).

```
def is_prime(num) :
    """Returns True iff 'num' is prime.

    Parameters:
        num (int): Integer value to be tested to see if it is prime.

    Return:
        bool: True if 'num' is prime. False otherwise.

    Preconditions:
        num > 1
    """
    if num <= 1 :
        raise ValueError("Input must be > 1")
    if num != int(num) :
        raise TypeError("Input must be an integer")

    i = 2
    while i < num :
        if num % i == 0 :
            return False
        i = i + 1
    return True
```

When we test the function on incorrect inputs, we see the function raises the appropriate errors:

```
>>> is_prime(-2)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    is_prime(-2)
  File "is_prime.py", line 9, in is_prime
    raise ValueError("Input must be > 1")
ValueError: Input must be > 1
>>> is_prime(3.14)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    is_prime(3.14)
  File "is_prime.py", line 11, in is_prime
    raise TypeError("Input must be an integer")
TypeError: Input must be an integer
```

`raise` Syntax

```
raise ExceptionName(args)
```

> **Semantics**
>
> The `raise` statement will cause an exception to be raised. If the `raise` statement is inside a `try` block, the interpreter will look for the appropriate `except` block to execute. Otherwise, the function will exit immediately, and the exception will **propagate**, exiting functions until it finds a `try`-`except` block. If there is no appropriate `try`-`except` block to handle the exception, the program will exit, and the Python interpreter will display an error message (`Traceback (most recent call last):`...)
>
> The *`ExceptionName`* should be a type of exception which is appropriate to the situation. The *`args`* can be any number of arguments, but is often a message to describe what caused the exception.
>
> `raise` statements are useful in the body of an `if` statement which checks if a value is invalid.