IFB104 — Building IT Systems
Topic 9 — How to Stop Programs Crashing

School of Computer Science

Semester 2, 2024

STUDENT VOICE SURVEY

Your feedback is important, no matter how small.

- Help to make positive changes to your learning experience.
- Each unit survey is an entry into the weekly draw to win $250. You can win more than once, so get in quick!
- A donation is made to QUT Guild Food Bank or Learning Potential Fund for every survey completed. So far we've raised $114,000.
- It only takes a few minutes to complete.

Not sure what comments to make?
Search HiQ for ‘providing professional feedback’ for ideas from QUT teachers.

QUT.TO/STUDENTVOICE

Housekeeping

- This is the last week with **new technical content** in IFB104
  - Next week you will **focus on Assignment 2B**
  - Week 13 will include **exam review**

This week

- So far we have learned how to build small IT systems, but our solutions have often been fragile
- We can make our programs more robust by applying a variety of techniques:
  - Efficient debugging
  - Defensive programming
  - Handling exceptions
  - Proving programs correct

Demonstration file: `adding_machine.py`

The need for robustness

- Users of IT systems have become accustomed to “crashes”
  - But would you accept a car that stalls and needs to be restarted every few kilometres?
  - So why do you accept this of your computer or mobile device?
- As IT professionals we should not be blasé about delivering faulty products to our clients

The Millenium Bug


WEEKLY WORLD
NEWS
THE COMPUTER CRASH OF THE MILLENNIUM!
JANUARY 1, 2000
IS THIS THE END?
ALL BANKS WILL FAIL!
FOOD SUPPLIES WILL BE DEPLETED!
THE STOCK MARKET WILL CRASH!
ELECTRICITY WILL BE CUT OFF!
VEHICLES USING COMPUTER CHIPS WILL STOP DEAD!
TELEPHONES WILL CEASE TO FUNCTION!
DEVASTATING WORLDWIDE DEPRESSION!
THE DAY THE EARTH STANDS STILL!


WEEKLY WORLD
NEWS
DOOMSDAY COUNTDOWN
THE FINAL DAYS
ALSO INSIDE ...
ARMAGEDDON
YEAR 2000 COMPUTER BUG will turn machine against man!
Hundreds of planes will fall out of the sky!
Cars will stop dead in their tracks!
Nuclear missiles will launch themselves!

Crisis

“Australia had a Y2K problem in two states when bus ticket validation machines failed to operate.”

https://web.archive.org/web/20040422221434/http://news.bbc.co.uk/2/hi/science/nature/590932.stm

Thanks to pre-emptive action from programmers, most code in critical systems was *robust* to the date rollover.

What happens if we don’t make code robust

theguardian

Thursday 24 November 2016 16.01 AEDT

Mars lander smashed into ground at 540km/h after misjudging its altitude

A tiny lander that crashed on Mars last month flew into the red planet at 540km/h (335mph) instead of gently gliding to a stop, after a computer misjudged its altitude, the European Space Agency has said.

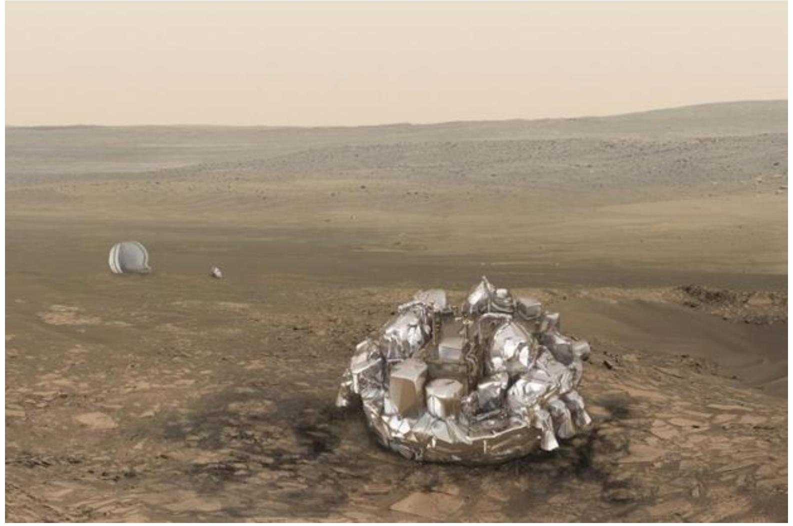
After trawling through vast amounts of data, the ESA said on Wednesday that while much of the mission went according to plan, a computer that measured the rotation of the lander hit a maximum reading, knocking other calculations off track.

“The erroneous information generated an estimated altitude that was negative – that is, below ground level,” the ESA said in a statement.

Integer overflow; maximum positive integer wrapped around to a negative number!

Error was allowed to propagate beyond the point it first occurred!

No sanity checking; altitudes when landing should not be negative!



Artist’s impression of the Schiaparelli Mars Lander smashed on the surface

A program that “works” but is very fragile

```
Enter barometer reading: 1
-- Retro rockets are off --
Enter barometer reading: 20
-- Retro rockets are off --
Enter barometer reading: 60
-- Retro rockets are off --
Enter barometer reading: 76
-- Retro rockets are firing --
Enter barometer reading: 88
-- Retro rockets are firing --
Enter barometer reading: 100
-- Retro rockets are off --
Houston, the Eagle has landed!
```

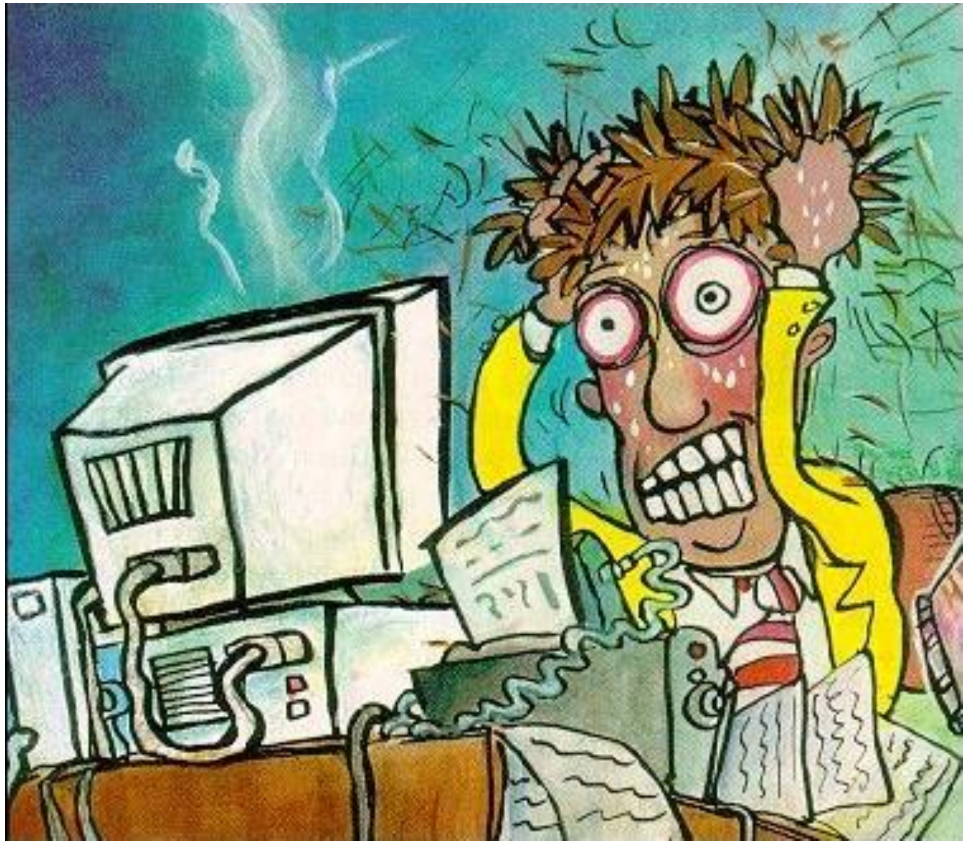
```
altitude = 1000000 # initialise with a big number

pressure_at_surface = 100 # constant

# Calculate altitude based on atmospheric
# pressure - higher pressure means lower altitude
def altimeter(barometer_reading):
    # Return the result
    return pressure_at_surface - barometer_reading

# Main program to decide when to fire the retros
while altitude != 0:
    # Read from the barometer (the user in this case!)
    air_pressure = int(input('Enter barometer reading: '))
    # Calculate the lander's altitude
    altitude = altimeter(air_pressure)
    # Decide whether or not the retros should be firing
    if altitude == 0 or altitude > 25:
        retros_off()
    else:
        retros_on()

# We made it!
print('Houston, the Eagle has landed!')
```

L1

Demonstration file: `mars_lander_V0.py`

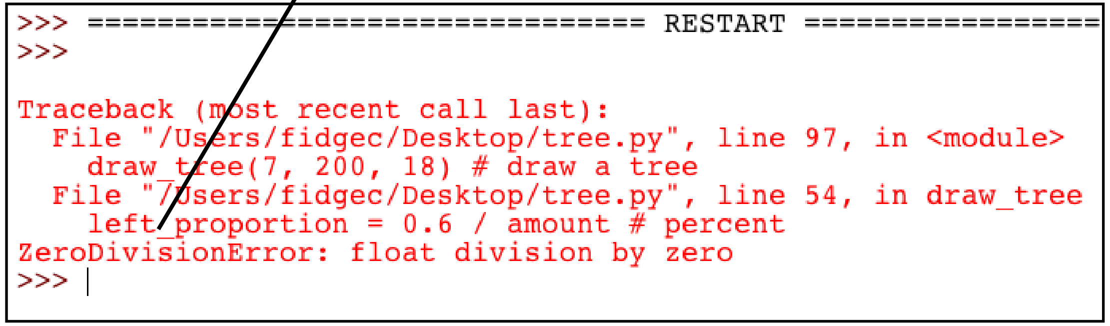
So why do IT systems fail?

- Mistakes made by the programmer:
  - Poorly designed algorithms
  - Coding errors
  - Failure to consider possible scenarios
- Problems outside the programmer's control:
  - The program being used in ways it's not designed for
  - Faults in the program's development or run-time environment
  - Malicious users and cyber attacks!
- Mistakes made by the person who commissioned the system
  - Omissions in the program's requirements specification



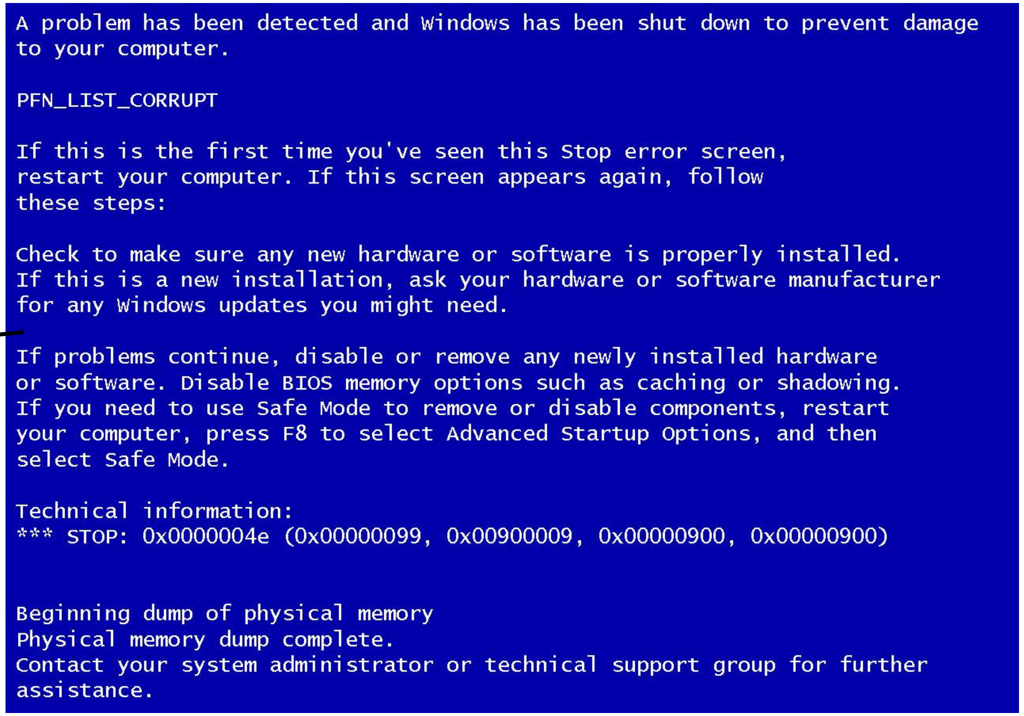
Different types of failures

A programming error caught by the development environment



A failure of an application caught by the operating system

A failure of the operating system itself



A failure of a failure!

How can your Python programs fail?

- By ‘crashing’, i.e., terminating abnormally:
  - Attempting to perform illegal/nonsensical operations

    ```
    TypeError: cannot
    concatenate 'str' and 'int'
    objects
    ```

  - Exhausting computational resources, e.g., memory space, number of concurrent threads, etc

    ```
    RuntimeError: maximum
    recursion depth exceeded
    ```

- By failing to run at all

    ```
    There's an error in your
    program: invalid syntax
    ```

- By running but not doing the right thing (semantic errors)

Part A — Errors of Omission

Building the wrong thing

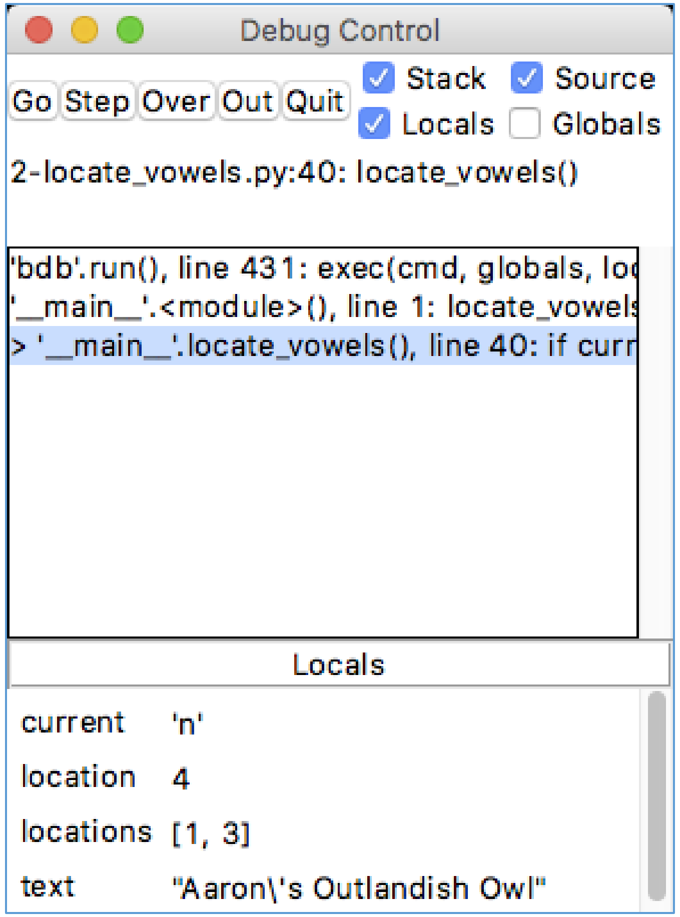
- Often when we deliver a new program to a client their response is, “That’s not what I wanted!”
- Possible reasons for this reaction:
  - We didn’t have the skills required to build the system requested
  - We didn’t understand what the client wanted
  - The client changed their mind
  - The client didn’t know what they wanted
- To solve this problem we need to become skilled at “requirements elicitation”
  - You will face this problem in later project units
- Some approaches:
  - Contractual requirements documents
  - Prototyping/mock ups
  - Agile development methodologies

Requirements elicitation can be difficult ...

Part B — Recap of Debugging Principles

Debugging your code

- Careful debugging is the first line of defence against mistakes made by the programmer
  - Merely adding a few calls to the `print` function to your code may be enough to help you diagnose a bug
  - In more complex situations use a tool like IDLE’s debugger to help trace variable values
    - Allows you to step through code
      - “Step” (goes through code)
      - “Over” (skip)
    - Can set breakpoints to pause at specific times

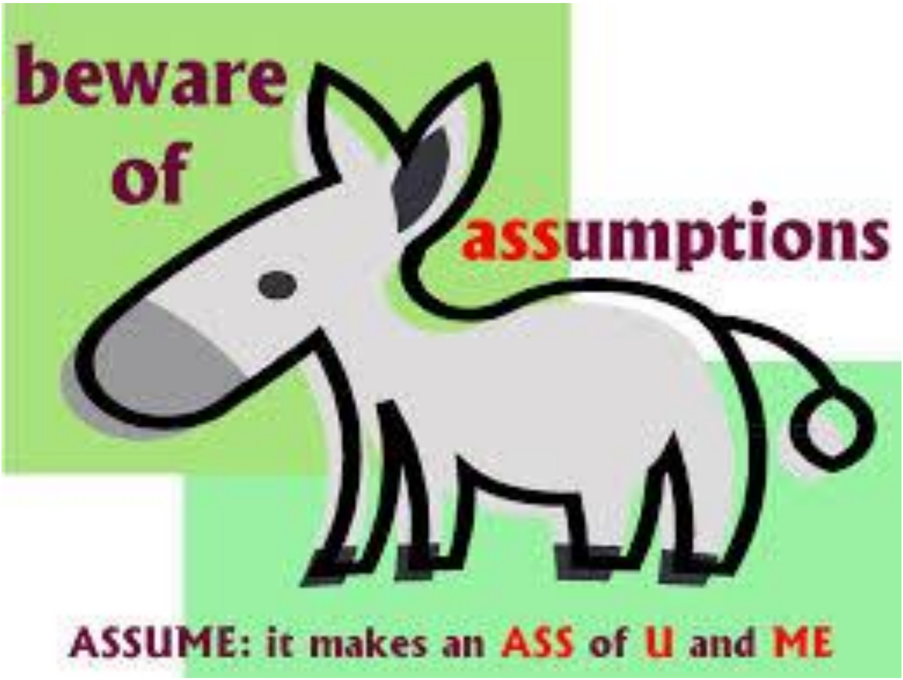


Demonstration file: `locate_vowels.py`

Part C — Defensive Programming

Assumptions and pre-conditions

- One cause of IT system failures is code being used in ways it's not designed for:
  - A function expecting an integer parameter is called with a string instead
  - A user asked to enter a number between 1 and 10 types "500"
- These situations are violations of the *assumptions* made during the code's development
  - In Computer Science these are also known as the code's *pre-conditions* for correct operation



Defensive programming

- Although we can’t prevent these situations, or even anticipate them all, we can make our programs more robust or *fault-tolerant* by practising *defensive programming*:
  - Check user inputs and function arguments for type and range validity
  - Catch run-time exceptions and check return codes
  - Make assertions about assumed program properties

Checking validity of external inputs

- The biggest threat to a program's calculations is receiving incorrect inputs
  - External inputs are beyond the program's control
- A "defensive" program will always check the validity of values received before relying on or using them
  - Here we use the regular expression **match** function to confirm the input is a well-formed integer
- If an invalid value is received the program may:
  - Request another value
  - Use a default value instead

```
# Main program to decide when to fire the retros
while altitude != 0:

    # Read a valid integer from the barometer
    raw_air_pressure = input('Enter barometer reading: ')
    while match('^-?[0-9]+$', raw_air_pressure) == None:
        raw_air_pressure = input('Re-enter barometer reading: ')
    air_pressure = int(raw_air_pressure)

    # Calculate the lander's altitude
    altitude = altimeter(air_pressure)
    # Decide whether or not the retros should be firing
    if altitude == 0 or altitude > 25:
        retros_off()
    else:
        retros_on()
```

```
Enter barometer reading: blob
Re-enter barometer reading: glop
Re-enter barometer reading: 45
-- Retro rockets are off --
```

Demonstration file: `mars_lander_V1.py`

Checking validity of parameters

- When we introduce parameters to our function definitions we make assumptions about their types and possible values
  - This is especially true in a language like Python where we don't have to declare the types of parameters
- A good coding habit is to check that the parameters provided really have the types and values we expect and to respond sensibly if they don't

```
# Calculate altitude based on atmospheric
# pressure - higher pressure means lower altitude
def altimeter(barometer_reading):

    # Sanity check - air pressure can't be negative
    if barometer_reading < 0:
        # Alert the space agency
        print('** Warning: Barometer failure! **')
        # Return the last known altitude as the best guess!
        return altitude

    # Return the result (if we make it this far!)
    return pressure_at_surface - barometer_reading
```

```
Enter barometer reading: 60
-- Retro rockets are off --
Enter barometer reading: 78
-- Retro rockets are firing --
Enter barometer reading: -10
** Warning: Barometer failure! **
-- Retro rockets are firing --
Enter barometer reading: 90
-- Retro rockets are firing --
```


Demonstration files: `is_square.py`, `mars_lander_V2.py`

Allow for unexpected values

- Another defensive programming technique is to anticipate that external input values may not always be precise
    - We should allow for slight errors in external inputs, especially when using floating point numbers

```
Enter barometer reading: 0
-- Retro rockets are off --
Enter barometer reading: 25
-- Retro rockets are off --
Enter barometer reading: 70
-- Retro rockets are off --
Enter barometer reading: 82
-- Retro rockets are firing --
Enter barometer reading: 101
-- Retro rockets are off --
Houston, the Eagle has landed!
```

If our Mars lander descends into a crater this loop may never terminate!

```
# Main program to decide when
# to fire the retros
while altitude != 0:
    ...
```

But this one always will

```
# Main program to decide when
# to fire the retros
while altitude > 0:
    ...
```

This condition would cause the retros to fire if the given altitude was negative

```
# Decide whether or not the retros
# should be firing
if altitude == 0 or altitude > 25:
    retros_off()
else:
    retros_on()
```

But this one turns them off if we appear to have descended below ground level

```
# Decide whether or not the retros
# should be firing
if altitude <= 0 or altitude > 25:
    retros_off()
else:
    retros_on()
```

Demonstration file: `mars_lander_V3.py`

Assertions

- *Assertions* are used to document things we believe to be true at certain points in the code:
  - They provide an easy way of defining expected *input values* (including function arguments) and *loop invariants*
  - They simplify program *debugging*, rather than inserting and removing calls to the `print` function
  - They are also the basis for program *correctness proofs*
- Assertions *do nothing*, as long as the program is well-behaved
  - But they produce a *fatal error* otherwise
  - They allow errors to be stopped <u>at their source</u>, rather than propagating to some other part of the program
- Sometimes assertions are used only during program development and are removed when the program is deployed (for run-time efficiency)

Assertions

- If an assertion is violated in a Python program it raises an **`AssertionError`** exception
  - This stops the program at the point of failure, to aid debugging

```
# Calculate altitude based on atmospheric
# pressure - higher pressure means lower altitude
def altimeter(barometer_reading):

    # Assertion to raise an exception if air pressure is negative
    assert barometer_reading >= 0, 'Barometer failure detected!'

    # Return the result (if we make it this far!)
    return pressure_at_surface - barometer_reading
```

```
Enter barometer reading: 34
-- Retro rockets are off --
Enter barometer reading: 56
-- Retro rockets are off --
Enter barometer reading: -2
Traceback (most recent call last):
AssertionError: Barometer failure detected!
```


Demonstration file: `separate_signs.py`, `mars_lander_V4.py`

Operator overloading

- Another way to make functions more robust is to allow them to handle data of different types … within reason!
  - For instance, it is usually sensible to allow a function that expects floating point numbers as arguments to accept integers
  - However, silently rounding floating point numbers to integers may not be safe
- Some programming languages, including Python, have many ‘overloaded’ operators, e.g., ‘**+**’ works for both numbers and sequences

```
>>> # Some integer arithmetic
>>> 4 + 5 * 2
14
>>> (4 + 5) * 2
18
>>> # Some string arithmetic
>>> 'Ha' + '!' * 3
'Ha!!!'
>>> ('Ha' + '! ') * 3
'Ha! Ha! Ha! '
>>> # Some list arithmetic
>>> ['x'] + ['y'] * 3
['x', 'y', 'y', 'y']
>>> (['x'] + ['y']) * 3
['x', 'y', 'x', 'y', 'x', 'y']
```


Demonstration file: `glue.py`

Part D — Exception Handling

What are exceptions?

- Earlier we saw examples of IT system “crashes”
  - Programming errors detected by the Interactive Development Environment
  - Application crashes detected by the run-time environment
  - Operating system crashes detected by the computer’s firmware kernel
- In general these kinds of problems are all examples of *unhandled exceptions*

- An exception is a situation where the running code cannot continue to run due to some serious problem
- Exceptions may be “raised” (or “thrown”) by the code itself or by its run-time environment
  - Thrown exceptions can be “caught” by the calling entity and resolved
  - Exceptions that are never caught will lead to crashes

Raising exceptions

- When the run-time system detects an error during program execution it “raises” (or “throws”) an *exception*
  - This usually results in the program terminating and an error message being displayed
- We can deliberately `raise` an exception ourselves if we believe an unrecoverable problem has occurred in our program
- Doing so alerts the calling code to the problem and gives it the opportunity to deal with the fault

```
# Given a record's diameter in inches,
# determine its speed in revolutions per
# minute
def rpm(diameter):
    if diameter == 10: # inches
        return 78 # revs per min
    if diameter == 7:
        return 45
    if diameter == 12:
        return 33
    # Something's wrong if we get
    # to this point!
    raise ValueError('Unknown record size')

print('Play your record at', rpm(8), 'rpms')
```

```
Traceback (most recent call last):
    raise ValueError('Unknown record size')
ValueError: Unknown record size
```


Demonstration file: `abbreviate.py`

Kinds of exceptions in Python

- There are many different kinds of exceptions
  - This allows us to tell not only that something went wrong but also to determine exactly what it was
- The built-in exceptions are organised in a hierarchy
  - This makes it possible to 'catch' a broad range of exceptions all at once or give special consideration to a specific one

```
BaseException
├── KeyboardInterrupt
├── Exception
│   ├── StandardError
│   │   ├── ArithmeticError
│   │   │   ├──
│   │   │   └── ...
│   │   ├── AssertionError
│   │   ├── LookupError
│   │   │   ├── IndexError
│   │   │   └── ...
│   │   ├── NameError
│   │   ├── SyntaxError
│   │   ├── TypeError
│   │   └── ...
│   └── ...
└── ...
```

Exception handlers

- An exception handler is a statement in two parts:
  - The 'normal' code that we want to execute
  - The 'exceptional' code that is executed if anything goes wrong with the normal code
- The flow of control is transferred from the normal code to the exception code as soon as an error occurs

```python
from math import sqrt
number = -4
try:
    print('The square root of', number, 'is:')
    print(sqrt(number))
except:
    print('Oops, something went wrong!')
```

```
The square root of -4 is:
Oops, something went wrong!
```

Exception handlers

- Using an *exception handler* we can respond gracefully to an exception that occurs while executing a group of statements

```
Enter barometer reading: 29
-- Retro rockets are off --
Enter barometer reading: 56
-- Retro rockets are off --
Enter barometer reading: -9
** Altimeter range error - Firing retros! **
-- Retro rockets are firing --
Enter barometer reading: 70
-- Retro rockets are off --
Enter barometer reading: gleep
** Altimeter type error - Firing retros! **
-- Retro rockets are firing --
Enter barometer reading: 97
-- Retro rockets are firing --
Enter barometer reading: 103
-- Retro rockets are off --
Houston, the Eagle has landed!
```

```
# Main program to decide when to fire the retros
while altitude > 0:

    # The default action for any kind of altimeter failure
    # is to assume we are at a low altitude - it's better
    # to waste fuel than crash!
    try:
        # Calculate the lander's altitude, if possible
        altitude = altimeter(int(input('Enter barometer reading: ')))
    except ValueError:
        print('** Altimeter type error - Firing retros! **')
        retros_on()
    except AssertionError:
        print('** Altimeter range error - Firing retros! **')
        retros_on()
    else:
        # Decide whether or not the retros should be firing
        if altitude <= 0 or altitude > 25:
            retros_off()
        else:
            retros_on()
```


Demonstration files: `show_title.py`, `adding_machine_gui.py`, `mars_lander_V5.py`

Some variations on the theme

- Several `except` clauses can be associated with the same `try` statement
- An exception handler can `raise` *another* exception if it decides that it can't handle the error adequately
  - The new exception will then propagate to the calling code

- Python allows an `else` clause on a `try` statement, which is executed if no exceptions occur
- Python allows a `finally` clause on a `try` statement, which is always executed whether an exception occurs or not

Developing code containing exception handling

- When developing a robust IT system it is best to keep ‘normal’ and ‘exceptional’ behaviours separate
    - Trying to think about both at the same time is confusing

- A typical approach is:

1. Develop the algorithm to satisfy its basic acceptance tests as if nothing will go wrong
2. After the basic algorithm is clear and complete, think about possible exception cases and add error-handling actions
3. Extend the acceptance tests to include the additional error-handling functionality

Demonstration files: `median.py`, `expression_history.py`, `text_file_miner.py`

When to use exception handling

- As its name suggests, ‘exception handling’ should be used for *exceptional* cases only, not as a routine way to control program execution
  - Only truly ‘fatal’ problems should result in exceptions
  - Raising an exception is justified when the ‘contract’ with the calling code (or user) is breached, e.g., when a function is called with arguments not allowed for by its requirements specification
- Raising exceptions is often safer than returning incorrect or ‘dummy’ values which may be confused for valid results
  - Using dummy values allows errors to propagate away from their source making the code harder to debug
- Code documentation should tell us which exceptions a function may raise, so we can react appropriately when we use it

Checking ‘return codes’

- As an alternative to exceptions, many scripting languages follow a convention whereby functions that perform some action (as a side effect) also return a ‘code’ indicating whether or not the action was successful
  - Sometimes the return code also indicates the nature of a failure
- If using such a language it is good programming practice to always check return codes and respond appropriately if a failure is indicated

```
less
NICE(1)                 BSD General Commands Manual                NICE(1)

NAME
     nice -- execute a utility with an altered scheduling priority

DIAGNOSTICS
     The nice utility shall exit with one of the following values:

     1-125    An error occurred in the nice utility.

     126      The utility was found but could not be invoked.

     127      The utility could not be found.

     Otherwise, the exit status of nice shall be that of utility.
:
```

Advanced topic: Creating your own exceptions

- As well as raising and handling built-in exceptions most programming languages allow you to define your own application-specific exceptions
- In Python this is done by creating a new exception *class*
  - See the Python documentation if you want to learn more about user-defined exceptions

Before next week ...

1. Complete this week’s workshop exercises
2. Work on Assessment 2B