Exceptions

Exceptions are events that can be raised and caught to modify the flow of control through a program.

They can be raised for errors, or used in other circumstances. They're meant to be used for unusual circumstances. They can also be used for system interrupts.

They are triggered automatically on errors, but we can also raise and intercept (catch) any exceptions in our own code.

There's a large list of exceptions already in Python.

Exception Handling

When writing code, the way exceptions normally work in Python is useful for debugging. Uncaught exceptions thrown filter up to the default exception handler.

However, when our code is being used, we need to handle the exceptions. At worst, we want the program to exit gracefully, and at best we want our program to be able to recover. E.g. if a user has entered a lot of data, we will want to save it (if possible) if our program has to exit.

We seek to minimise data loss and clean up connections. If we can't try again, we want a stable situation. If we can't have that, we want a graceful exit.

Key Mechanisms

- `try` / `except` – this construct catches and recovers from exceptions
- `try` / `finally` – provides a block that is always executed, even if an exception is thrown
- `raise` – allows us to trigger exceptions from within our code

`try`/`except`

We intercept exceptions with the `try`/`except` construct. We wrap operations that are likely to throw an exception in the `try` block, and provide error handling in the `except` block.

This means that any code after the `try`/`except` structure can still run even if exceptions have been thrown.

You can catch multiple exceptions with multiple `except` blocks. You can also use `except Exception` as a general exception handler – it will catch any exception thrown (except for some, e.g. system interrupt) that's not already handled.

The Google Style Guide recommends against catchall exceptions (unless at the outermost level of error catching where some error is being shown to the user), and says you should only catch specific exceptions. There are some situations where you may choose to use a catchall. We'll see more of this with GUIs.

You should never use a block like this:

```
…
except:
    …
```

This will catch some exceptions that aren't errors, that Python needs to function properly, and can cause unusual effects in your code.

Cleaning Up

We use `try`/(`except`/)`finally` to have some cleanup code that will run whether or not an exception was thrown.

Raising Exceptions

We use the raise construct to raise an exception. We can write our own exceptions classes, which we will see later.

Here's the syntax:

```
raise TypeError("Tried to set name with an illegal type")
```

The string is the message that will be shown by the default exception handler.

We should also document exceptions that our code throws, especially in our class methods. this allows third parties to see what exceptions need to be dealt with when they use our class.

Example

```
class Student(object):

    def __init__(self, name, studentid):

        if not isinstance(name, str):
            raise TypeError("Name is wrong type")
        if not isinstance(studentid, str):
            raise TypeError("Student ID is wrong type")

        if name == '':
            raise ValueError("Name must be non-empty")

        self._name = name
        self._studentid = studentid

def main():
    tempname = 12345
    tempid = ""

    try:
        student = Student(tempname, tempid)
    except TypeError:
        #Prompt the user for fresh values with feedback
        print("Some variable had the wrong type.")
    except ValueError:
        print("Some variable had an incorrect value.")

if __name__ == "__main__":
    main()
```