COMP110: Principles of Computing

8: Exceptions and debugging

Learning outcomes

- **Explain** the usefulness of exceptions and assertions in writing robust software
- **Write** Python programs that throw and catch exceptions
- **Use** the PyCharm debugger to trace the execution of programs

Reading

E. Dijkstra, 1968. Go To Statement Considered Harmful. *Communications of the ACM,* 11(3):147–148.

Modular program design

Modular program design

- Writing a piece of software as a **single long source file** is generally a bad idea
- Better to split the program into several self-contained files, i.e. **modules**
- Makes code easier to **navigate**
- Allows related classes and functions to be **grouped** together
- Modules can often be **reused** between programs
- In compiled languages, can allow for faster iteration via **incremental recompilation**

Modules in Python

- **Every** Python source file is a **module**
- Can be loaded using the **`import`** statement, just like built-in modules

mymodule.py

```
def test():
    print "Hello world!"
```

program.py

```
import mymodule

mymodule.test()
```

Modules vs programs

- How to tell if a Python file is a program or a module?
- Every Python file can be used as both, but you can check at runtime:

```
if __name__ == "__main__":
    print "I am a program"
else:
    print "I am an imported module"
```

- `__name__` is a built-in variable (note the double underscores!)

Exceptions

Exceptions

- You've all seen them already...

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

- Some operations in a program can fail
- Exceptions are a way of signalling that failure

Raising exceptions

```
def factorial(n):
    if n < 0:
        raise ValueError("Argument must be positive")
```

- `ValueError` is a built-in class
- Its initialiser takes one argument: a human-readable string describing the error
- The exception is an **instance** of the `ValueError` class

Built-in exception types

https://docs.python.org/2/library/exceptions.html

Custom exception types

```
class UnreticulatedSplineError(Exception):
    def __init__(self, message):
            Exception.__init__(self, message)


...
if not spline.is_reticulated():
    raise UnreticulatedSplineError("Spline must be ↩
        reticulated")
```

- Inherit from built-in class `Exception`
- Just like any other class — can contain any required fields and methods

Catching exceptions

```
try:
    input_file = open(filename, "rt")
except IOError:
    print "Failed to open file, please select a ↩
        different one"
```

- If anything inside the **`try`** block throws an `IOError` (or a subclass of `IOError`), the **`except`** block executes
- A **`try`** block can have several **`except`** blocks for different exception types
- An uncaught exception kills the program

Exceptions and control flow

- Raising exception transfers control to the innermost matching `except` handler
- This can result in breaking out of loops and functions
- This can be powerful in the hands of a good programmer...
- ... or confusing in the hands of a bad one

Types of exceptions

There are two types of exceptions...

- Those intended to catch **runtime errors** (e.g. missing files, insufficient resources, invalid input, ...)
  - Good software should **catch** and **recover** from these
- Those intended to catch **programmer errors** (e.g. type mismatch, index out of bounds, divide by zero, ...)
  - Generally best to let these crash the program so that the programmer can notice and fix them
  - A commercially released program might catch them to allow the user to submit a bug report to the developer

Assertions

- `AssertionError` is a special exception type for catching programmer errors
- Can be raised with the following syntax:

```
assert n > 0, "n must be positive"
```

- The exception is raised if the condition is `False`
- `assert` should **only** be used to catch programmer errors, therefore you should **never** try to handle `AssertionError` in a **`try ... except`** block
- In some programming languages, assertions are stripped out of "Release" builds — so avoid assertions with side-effects!

More options with try blocks

```
try:
    input_file = open(filename, "rt")
except IOError as err:
    print "File error:", err
else:
    print "This only executes if the try block did not ↩
         raise an exception"
finally:
    print "This executes whether or not the try block  ↩
        raised an exception, even if there were  ↩
        uncaught exceptions"
```

It's easier to ask forgiveness than permission

I.e. it's better to catch exceptions than to use **`if`** statements to avoid them

```
try:
    x = list[index]
except IndexError:
    x = None
```

This is "more Pythonic" than this:

```
if index >= 0 and index < len(list):
    x = list[index]
else:
    x = None
```

The debugger

“Why doesn’t my program work?”

There are several ways to try and figure out what a program is doing:

- Add **print statements** and/or **logging**
- **Change** or **comment out** parts and see what the effect is
- **Trace** the execution of the program with pen and paper

But there’s a better way...

What is a debugger?

A **debugger** is a set of tools built into an IDE, typically allowing the programmer to:

- **Pause** a running program
  - Manually (pause button)
  - "Run to cursor"
  - Upon reaching a **breakpoint**, which may be **conditional**
  - When an **exception** is raised and not handled
- **Step** through the execution of the program, line by line
- **Inspect** the values of variables, and sometimes **change** them
- View the **call stack**

Debugging in PyCharm

- **Run** $\rightarrow$ **Debug** or the  button
- To set a breakpoint, click in the left margin of the code — a red dot should appear
- Stepping through code:
  - **Step over**: run to the next statement in the current function
  - **Step into**: if the current statement is a function call, step into that function
  - **Step out**: run until the current function returns

Debugging in PyCharm

(demo)

Worksheet D