1811/2807/7001ICT
Programming Principles

School of Information and Communication Technology
Griffith University

Trimester 1, 2024

19 Errors and Exceptions

This section explores the kinds of errors we find in programs, and introduces the `try` – `except` control statement.

19.1 Kinds of errors

These kinds of errors are found in programs:

Syntax errors – any problems which are detected as the program is parsed.

These are errors involving the arrangement of symbols in the source file.

Logical/semantic errors – errors made by the programmer that cause the program to not mean what the programmer intended or to perform as required.

Unavoidable run-time errors – problems that are detected at run time, but can not be avoided by better programming, including: out of disk space; a file doesn't exist; or a lost network connection.

19.2 Error handling in programs

It is good to write programs that can cope with all reasonably expected error situations, but it is not always good to clutter programs with unnecessary precautionary checks.

19.2.1 Defensive programming

Here is a program with a problem.

```
# script: Middle1.py

def middleChar(s):
   """Returns the character at the middle of s."""
   return s[len(s) // 2]

print(middleChar(None))
```

```
$ python3 Middle1.py
Traceback (most recent call last):
  File "Middle1.py", line 7, in <module>
    print(middleChar(None))
  File "Middle1.py", line 5, in middleChar
    return s[len(s) // 2]
TypeError: object of type 'NoneType' has no len()
$
```

This error is caused because we tried to find the length of `None`.

The interpreter is very nice to us and tells us what line the error occured on, in which function and from which line was that called.

What code is *actually* at fault?

Same function, different problem.

```
# script: Middle2.py

def middleChar(s):
   """Returns the character at the middle of s."""
   return s[len(s) // 2]

print(middleChar(""))
```

```
$ python3 Middle2.py
Traceback (most recent call last):
  File "Middle2.py", line 7, in <module>
    print(middleChar(""))
  File "Middle2.py", line 5, in middleChar
    return s[len(s) // 2]
IndexError: string index out of range
$
```

A *defensive* programmer might get a bit nervous about the `middleChar` function, and try to make it safe by checking that the argument is valid...

```
# script: Middle3.py

def middleChar(s):
   """Returns the character at the middle of s."""
   if s != None and len(s) > 0:
      return s[len(s) // 2]
   else
      # What should I do here?
```

But the function works perfectly for sensible inputs, so why not just demand that the caller get it right!

19.2.2 Programming by contract

A contract is a document that describes an agreement between two parties, obliging each to do something, to the benefit of both.

"If you build this shed for me, I'll give you $500."

If you don't build the shed, I'm not obliged to pay.

We write a "contract" as preconditions and postconditions for a method.

precondition: If you give me a string with at least one character in it,
postcondition: I'll find the middle one for you.

If you don't meet the preconditions of a function, then it's not my fault that it doesn't work.

Pre- and postconditions are good places to be as precise as possible about what a method requires and does.

This clearly documents the pre- and postconditions.

```
# script: Middle4.py

def middleChar(s):
   """Returns the character at the middle of s.

      Precondition: len(s) > 0
      Postcondition: Returns the character at the
      least position closest to the middle of s."""
   return s[len(s) // 2]
```

That this function dies horribly for invalid inputs is a good thing.

It motivates us to fix the errors in calls to `middleChar`.

When all such errors are fixed, there is no point in having defensive error checking code.

This example has been a case where a logical error causes a run-time error.

We handle logical errors by testing for them, then fixing them.

19.2.3 Unavoidable, unexpected errors

There are still those run-time errors caused by unexpected events from which a program could recover, such as missing files.

Should we be defensive?

Yes, but we should not let the code that handles rare problems clutter otherwise clear code.

Python lets us separate the code that does the job, from the code that handles the rare problems using the `try` – `except` statements.

19.2.4 try – except

Template:

```
try:
   risky actions(s)
except:
   recovery actions(s)
```

The *risky action(s)* are executed. If one fails, raising an exception, control jumps to the *recovery action(s)*.

If the *risky action(s)* succeed, the *recovery action(s)* are skipped.

Example:

```
# file: zero.py
# divide by zero

try:
   print(1 // 0)
except:
   print("Don't do that, dummy.")
```

```
$ python3 zero.py
Don't do that, dummy.
$
```

Don’t overuse exception handling.

That was a bad example.

The potential for division by zero can be detected with a simple selection.

Exception handling is really for handing errors like trying to read from a file that doesn’t exist.

Look for better examples in the next section, *Files*.

Section summary

This section covered:

- the categories of errors, syntax, logical, run-time;
- defensive programming;
- programming by contract; and
- the `try` – `except` statement for handling exceptions.