

Defensive programming

Week 3, day 2

Writing correct programs

- Handle errors / bugs in 2 ways:
 - Reactive: wait for a problem to occur
 - Defensive: catch problems as soon as they occur
- Assertions & exceptions

Coding for failure

```
def getSeqFromFasta (fileName):  
    """  
    Return list of seqs from file.  
    """  
    import os  
    if (os.exists (fileName)):  
        theInFile = open (fileName, 'r')  
        if (theInFile.open):  
            theCurrLine = theInFile.readline()  
            if (theCurrLine.startswith(">")):  
                # etcetera, etectera  
                # for 50-95% of program
```

Assertions

```
def getSeqFromFasta (fileName):  
    """  
    Return list of seqs from file.  
    """  
    import os  
    assert (os.exists (fileName)), "the file doesn't exist"  
    theInFile = open (fileName, 'r')  
    assert (theInFile.open), "can't open the file"  
    theCurrLine = theInFile.readline()  
    assert (theCurrLine.startswith(">"), """  
        no seq start, maybe file malformed""")
```

assert

- `assert`
- “This must be true”
- Check for failure, raises an error with given message

```
def getSeqFromFasta (fileName):  
    """  
    Return list of seqs from file.  
    """  
    import os  
    assert (os.exists (fileName)), \  
           "the file doesn't exist"  
    theInFile = open (fileName, 'r')  
    assert (theInFile.open), \  
           "can't open the file"  
    [...]
```

Calling assert

- `assert test [, data]`
- `test` should not
 - Be expensive to calculate
 - Mutate any objects
- `data` is usually a string. Usually.

Pre- & post-conditions

- What has to be true:
 - before the function starts
 - after it finishes

```
def getSeqFromFasta (fileName):  
    """  
    Return list of seqs from file.  
    """  
    # Preconditions:  
    import os  
    assert (os.exists (fileName)), \  
           "the file doesn't exist"  
    # Main:  
  
    [...]  
  
    # Postconditions:  
    assert len (theSeqs), "no seqs"  
    return theSeqs
```

Exceptions

- Error
- A more expressive mechanism
- A way of reporting errors and handling them so errors are non-fatal
- We raise or throw an exception

Raising exceptions

```
def getSeqFromFasta (fileName):  
    """  
    Return list of seqs from file.  
    """  
    # Preconditions:  
    if (not os.exists (fileName)):  
        raise RuntimeError, """'%s'  
        doesn't exist""" % fileName
```

Raising exceptions (2)

- *raise exceptionClass [, data]*
- Produces an exception of that class with attached data
- can get data out of error by converting to string, e.g. `str(myErr), "%s" % myErr`
- Breaks execution flow ...

Assertions are exceptions

```
# the same
assert (inFile.open), "file not open"

if (not inFile.open):
    raise AssertionError, "file not open"

# the same
assert cond, data

if not cond:
    raise AssertionError, data
```

Handling exceptions

```
try:
    if (not os.exists (fileName)):
        raise RuntimeError, "file doesn't exist"
    if (not inFile.open):
        raise AssertionError, "file not open"
except:
    print "An error occurred with the file"
```

Try-except general form

```
try:  
    # something that may raise error  
except [exceptionclass [, error]]:  
    # do something with error  
except [exceptionclass [, error]]:  
    # do something with error  
except [exceptionclass [, error]]:  
    # etcetera
```

Try-except examples

```
try:
    if (not os.exists (fileName)):
        raise RuntimeError, "file doesn't exist"
    if (not inFile.open):
        raise AssertionError, "file not open"
except RuntimeError, e:
    print "A runtime error occurred: %s" % e
except AssertionError, e:
    print "An assertion error occurred: %s" % e
except:
    print "Something unexpected happened!"
```

Try-except examples 2

```
except RuntimeError:  
    # catch all RuntimeError objects or subclasses  
  
except RuntimeError, e:  
    # ... and make e refer to the caught error  
  
except:  
    # catch everything
```

```

Exception(*)
|
+-- StandardError(*)
|
|   +-- KeyboardInterrupt
|   +-- ImportError
|   +-- EnvironmentError(*)
|       |
|       +-- IOError
|       +-- OSError(*)
|
+-- EOFError
+-- RuntimeError
|   |
|   +-- NotImplementedError(*)
|
+-- NameError
+-- AttributeError
+-- SyntaxError
+-- TypeError
+-- AssertionError
+-- LookupError(*)
|   |
|   +-- IndexError
|   +-- KeyError
|
+-- ArithmeticError(*)
|   |
|   +-- OverflowError
|   +-- ZeroDivisionError
|   +-- FloatingPointError
|
+-- ValueError
+-- SystemError
+-- MemoryError

```

(Some of)
The error
classes

Try-except examples 3

```
# move from specific to general

except RuntimeError, e:
    # catch all RuntimeError objects or subclasses
except StandardError, e:
    # catch all StandardError objects or subclasses
except:
    # catch everything
```

Custom exceptions

```
import exceptions

class ParseError (exceptions.Exception):
    def __init__ (self, msg, file, line):
        exceptions.Exception.__init__ (self, msg)
        self.file = file
        self.line = line

    def __str__ (self):
        return exceptions.Exception.__str__ (self) + \
            "%s:%s" % (self.file, self.line)
```

Exceptions walk up the call stack

```
try:  
    theSeqs = parsePhylipData()  
except:  
    print "bad data!"
```

```
def parsePhylipData():  
    openInfile()  
    [...]
```

```
def openInfile():  
    getFileName()  
    [...]
```

```
def getFileName():  
    assert nameIsValid()
```

Making it someone elses problem

```
except RuntimeError, e:  
    # do something  
except:  
    # re-raise errors  
    raise
```

Trying to import

```
try:
    import Numeric
    numLib = "numeric"
except:
    # numeric not available
    import numarray
    numLib = "numarray"
```

Errors in Abstract base classes

```
class abstractDatabaseEntry:
    # blah, blah ...

    def getDatabaseReference (self):
        raise NotImplementedError, """
            must over-ride in base class!!!"""
```

When to test what

- Assertions
 - Initially for the developer
 - Can leave in released/final code
- Exceptions
 - For exceptional conditions
 - Runtime

When to test what (2)

- Use assertions, postconditions & preconditions to check your logic
- Leave assertions in the final version
- Distrust anything that comes from the outside & throw exceptions
- Handle all exceptions

Exceptions are not flow control

- Exceptions seem like a good way to escape from nested function calls
- Almost always a bad idea:
 - What about other exceptions?
 - Poor readability