What is an exception?

- What happens when procedure execution hits an unexpected condition?
  - Trying to access beyond the limits of a list will raise an `IndexError`
    - `Test = [1,2,3]`
    - `Test[4]`
  - Trying to convert an inappropriate type will raise a `TypeError`
    - `Int(Test)`
  - Referencing a non-existing variable will raise a `NameError`
    - `a`
  - Mixing data types without appropriate coercion will raise a `TypeError`
    - `'a'/4`
- These are **exceptions –**exceptions to what was expected

What to do with exceptions?

- What to do when procedure execution is stymied by an error condition?
  - Fail silently: substitute default values, continue
    - Bad idea! User gets no indication results may be suspect
  - Return an "error" value
    - What value to chose? `None?`
    - Callers must include code to check for this special value and deal with consequences ➔ cascade of error values up the call tree
  - Stop execution, signal error condition
    - In Python: **raise an exception**

```
raise Exception("descriptive string")
```

Dealing with exceptions

- Python code can provide handlers for exceptions

```
try:
   f = open('grades.txt')
    # … code to read and process grades
except:
   raise Exception("Can't open grades file")
```

- Exceptions raised by statements in body of `try` are handled by the `except` statement and execution continues with the body of the `except` statement

Handling specific exceptions

- Usually the handler is only meant to deal with a particular type of exception. Sometimes we need to clean up before continuing.

```
try:
   f = open('grades.txt')
    # … code to read and process grades
except IOError,e:
   print "Can't open grades file: " + str(e)
   sys.exit(0)
except ArithmeticError,e:
   raise ValueError("Bug in grades calculation " +
                    str(e))
```

Types of exceptions

- Already seen common error types:
  - SyntaxError: Python can’t parse program
  - NameError: local or global name not found
  - AttributeError: attribute reference fails
  - TypeError: operand doesn’t have correct type
  - ValueError: operand type okay, but value is illegal
  - IOError: IO system reports malfunction (e.g. file not found)

Other extensions to `try`

- `else`:
  - Body of this clause is executed when execution of associated `try` body completes with no exceptions
- finally:
  - Body of this clause is always executed after `try`, `else` and `except` clauses, even if they raised another error or executed a `break`, `continue` or `return`
  - Useful for clean-up code that should be run no matter what else happened (e.g. close file)

An example

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError, e:
        print "division by zero! " + str(e)
    else:
        print "result is", result
    finally:
        print "executing finally clause"
```

An example, revised

```
def divideNew(x, y):
    try:
        result = x / y
    except ZeroDivisionError, e:
        print "division by zero! " + str(e)
    except TypeError:
        divideNew(int(x), int(y))
    else:
        print "result is", result
    finally:
        print "executing finally clause"
```

An example of exceptions

- Here is an example of how we can use exceptions to handle unexpected situations in code
- Assume we are given a class list for a subject: each entry is a list of two parts – a list of first and last name for a student, and a list of grades on assignments
- We want to create a new subject list, with name, grades, and a weighted average

A simple start

```
def getSubjectStats(subject, weights):
    return [[elt[0], elt[1], avg(elt[1], weights)]
            for elt in subject]

def dotProduct(a,b):
    result = 0.0
    for i in range(len(a)):
        result += a[i]*b[i]
    return result

def avg(grades, weights):
    return dotProduct(grades, weights)/len(grades)
```

An error if no grades for a student

- If we run this on a list of students, one or more of which don't actually have any grades, we get an error:

```
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    getSubjectStats(test, weights)
  File
"/Users/ericgrimson/Documents/6.00x/subjectCode
.py", line 3, in getSubjectStats
    for elt in subject]
  File
"/Users/ericgrimson/Documents/6.00x/subjectCode
.py", line 12, in avg
    return dotProduct(grades,
weights)/len(grades)
ZeroDivisionError: float division by zero
```

Let's flag the error

```
def avg(grades, weights):
    try:
        return dotProduct(grades, weights)/len(grades)
    except ZeroDivisionError:
        print 'no grades data'
```

Running on some test data yields

```
>>> getSubjectStats(test, weights)
no grades data
[[['fred', 'flintstone'], [10.0, 5.0, 85.0], 15.5],
[['barney', 'rubble'], [10.0, 8.0, 74.0],
13.866666666666667], [['wilma', 'flintstone'], [8.0,
10.0, 96.0], 17.466666666666665], [['dino'], [], None]]
```

Note that last entry now has a 'None' object for the average grade

Or we could change policy

- Suppose we decide that a student with no grades is getting a zero in the class:

```
def avg(grades, weights):
    try:
        return dotProduct(grades, weights)/len(grades)
    except ZeroDivisionError:
        print 'no grades data'
        return 0.0
```

```
>>> getSubjectStats(test, weights)
no grades data
[[['fred', 'flintstone'], [10.0, 5.0, 85.0], 15.5],
[['barney', 'rubble'], [10.0, 8.0, 74.0],
13.866666666666667], [['wilma', 'flintstone'], [8.0,
10.0, 96.0], 17.466666666666665], [['dino'], [], 0.0]]
```

We can handle multiple exceptions

- Suppose some grades are “letter” grades. We can convert them using

```
def convertLetterGrade(grade):
    if type(grade) == int:
        return grade
    elif grade == 'A':
        return 90.0
    elif grade == 'B':
        return 80.0
    elif grade == 'C':
        return 70.0
    elif grade == 'D':
        return 60.0
    else:
        return 50.0
```

We can handle multiple exceptions

```
def avg(grades, weights):
    try:
        return dotProduct(grades, weights)/len(grades)
    except ZeroDivisionError:
        print 'no grades data'
        return 0.0
    except TypeError:
        newgr = [convertLetterGrade(elt) for elt in grades]
        return dotProduct(newgr, weights)/len(newgr)
```

We can handle multiple exceptions

```
>>> getSubjectStats(test1, weights1)
no grades data
[['fred', 'flintstone'], [10.0, 5.0, 85.0,
'D'], 10.0], [['barney', 'rubble'], [10.0,
8.0, 74.0, 'B'], 11.25], [['wilma',
'flintstone'], [8.0, 10.0, 96.0, 'A'],
11.875], [['dino'], [], 0.0]]
```

Exceptions as flow of control

- In traditional programming languages, one deals with errors by having functions return special values
- Any other code invoking a function has to check whether ‘error value’ was returned
- In Python, can just raise an exception when unable to produce a result consistent with function’s specification
  - `raise exceptionName(arguments)`

Example

```
def getRatios(v1, v2):
    """Assumes: v1 and v2 are lists of equal length of numbers
       ReturnsL a list containing the meaningful values of
           v1[i]/v2[i]"""
    ratios = []
    for index in range(len(v1)):
        try:
            ratios.append(v1[index]/float(v2[index]))
        except ZeroDivisionError:
            ratios.append(float('NaN')) #NaN = Not a Number
        except:
            raise ValueError('getRatios called with bad arg')
    return ratios
```

Using the example

```
try:
   print getRatios([1.0,2.0,7.0,6.0],
                   [1.0,2.0,0.0,3.0])
   print getRatios([],[])
   print getRatios([1.0,2.0], [3.0])
except ValueError, msg:
   print msg
```

```
[1.0, 1.0, nan, 2.0]
[]
getRatios called with bad argument
```

Compare to traditional code

```
def getRatios(v1, v2):
    """Assumes: v1 and v2 are lists of equal length of numbers
       ReturnsL a list containing the meaningful values of
           v1[i]/v2[i]"""
    ratios = []
    if len(v1) != len(v2):
        raise ValueError('getRatios called with bad arg')
    for index in range(len(v1)):
        v1Elt = v1[index]
        v2Elt = v2[index]
        if (type(v1Elt) not in (int, float))\
           or (type(v2Elt) not in (int, float)):
            raise ValueError('getRatios called with bad arg')
        if v2Elt == 0.0:
            ratios.append(float('NaN')) #NaN = Not a Number
        else:
            ratios.append(v1Elt/v2Elt))
    return ratios
```

Compare to traditional code

- Harder to read, and thus to maintain or modify
- Less efficient
- Easier to think about processing on data structure abstractly, with exceptions to deal with unusual or unexpected cases

Assertions

- If we simply want to be sure that assumptions on state of computation are as expected, we can use an `assert` statement
- We can't control response, but will raise an `AssertionError` exception if this happens
- This is good defensive programming

Example

```
def avg(grades, weights):
    assert not len(grades) == 0, 'no grades data'
    newgr = [convertLetterGrade(elt) for elt in grades]
    return dotProduct(newgr, weights)/len(newgr)
```

This will raise an `AssertionError` if it is given an empty list for grades, but otherwise will run properly

- error will print out information `'no grades data'` as part of process

Assertions as defensive programming

- While assertions don't allow a programmer to control response to unexpected conditions, they are a great method for ensuring that execution halts whenever an expected condition is not met
- Typically used to check inputs to procedures, but can be used anywhere
- Can make it easier to locate a source of a bug

Extending use of assertions

- While pre-conditions on inputs are valuable to check, can also apply post-conditions on outputs before proceeding to next stage

Example, extended

```
def avg(grades, weights):
    assert not len(grades) == 0, 'no grades data'
    assert len(grades) == len(weights), 'wrong number grades'
    newgr = [convertLetterGrade(elt) for elt in grades]
    result = dotProduct(newgr, weights)/len(newgr)
    assert 0.0 <= result <= 100.0
    return result
```

Example, extended

- Slight loss of efficiency
- Defensive programming:
  - by checking pre- and post-conditions on inputs and output, avoid propagating bad values

Where to use assertions?

- Goal is to spot bugs early, and make clear where they happened
  - Easier to debug when catch at first point of contact, instead of trying to trace down later
- Not to be used in place of testing, but as a supplement to testing
- Should probably rely on raising exceptions if users supplies bad data input, and use assertions for:
  - Checking types of arguments or values
  - Checking that invariants on data structures are met
  - Checking constraints on return values
  - Checking for violations of constraints on procedure (e.g. no duplicates in a list)