EXCEPTIONS, ASSERTIONS

EXCEPTIONS AND ASSERTIONS

- what happens when procedure execution hits an **unexpected condition**?
- get an **exception**… to what was expected
  - trying to access beyond list limits

```
test = [1,7,4]
test[4]
```
→ `IndexError`

  - trying to convert an inappropriate type

```
int(test)
```
→ `TypeError`

  - referencing a non-existing variable

```
a
```
→ `NameError`

  - mixing data types without coercion

```
'a'/4
```
→ `TypeError`

OTHER 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)

WHAT TO DO WITH EXCEPTIONS?

- what to do when encounter an error?
- **fail silently**:
  - substitute default values or just continue
  - bad idea! user gets no warning
- return an **"error" value**
  - what value to choose?
  - complicates code having to check for a special value
- 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:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
    print ("Okay")
except:
    print("Bug in user input.")
print(”Outside")
```

- exceptions **raised** by any statement in body of **`try`** are **handled** by the **`except`** statement and execution continues after the body of the `except` statement

HANDLING SPECIFIC EXCEPTIONS

- have **separate `except` clauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

only execute if these errors come up

for all other errors

OTHER EXCEPTIONS

- `else`:
  - body of this is executed when execution of associated `try` body **completes with no exceptions**
- `finally`:
  - body of this 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 a file)

EXAMPLE EXCEPTION USAGE

```
while True:
    try:
        n = input("Please enter an integ
        n = int(n)
        break
    except ValueError:
        print(”Input not an integer; try again”)
print(”Correct input of an integer!”)
```

Loop only exits when correct type of input provided

Only prints message if exception raised

EXAMPLE: CONTROL INPUT

```
data = []
file_name = input("Provide a name of a file of data ")

try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    for new in fh:
        if new != '\n':
            addIt = new[:-1].split(',') #remove trailir
            data.append(addIt)
finally:
    fh.close() # close file even if fail
```

Jump out if no file of that name

Close file in either case

EXAMPLE: CONTROL INPUT

- appears to correct read in data, and convert to a list of lists
- now suppose we want to restructure this into a list of names and a list of grades for each entry in the overall list

EXAMPLE: CONTROL INPUT

```
data = []
file_name = input("Provide a name of a file of data ")

try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    for new in fh:
        if new != '\n':
            addIt = new[:-1].split(',') #remove trailing \n
            data.append(addIt)
finally:
    fh.close() # close file even if fail

gradesData = []
if data:
    for student in data:
        try:
            gradesData.append([student[0:2], [student[2]]])
        except IndexError:
            gradesData.append([student[0:2], []])
```

Handle case of no grade;
But assumes two names!

EXAMPLE: CONTROL INPUT

- works okay if have standard form, including case of no grade
- but fails if names are not two parts long

EXAMPLE: CONTROL INPUT

```
data = []
file_name = input("Provide a name of a file of data ")

try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    for new in fh:
        if new != '\n':
            addIt = new[:-1].split(',') #remove trailing \n
            data.append(addIt)
finally:
    fh.close() # close file even if fail

gradesData = []
if data:
    for student in data:
        try:
            name = student[0:-1]
            grades = int(student[-1])
            gradesData.append([name, [grades]])
        except ValueError:
            gradesData.append([student[:], []])
```

Handle case of no grade;
Now allows for multiple names!

EXCEPTIONS AS CONTROL FLOW

- don't return special values when an error occurred and then check whether 'error value' was returned
- instead, **raise an exception** when unable to produce a result consistent with function's specification

```
raise <exceptionName>(<arguments>)
```

```
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

typically a string with a message

EXAMPLE: RAISING AN EXCEPTION

```
def get_ratios(L1, L2):
    """ Assumes: L1 and L2 are lists of equal length of numbers
        Returns: a list containing L1[i]/L2[i] """
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/float(L2[index]))
        except ZeroDivisionError:
            ratios.append(float('NaN')) #NaN = Not a Number
        except:
            raise ValueError('get_ratios called with bad arg')
    return ratios
```

manage flow of program by raising own error

EXAMPLE OF EXCEPTIONS

- 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
  - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
[['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]
```

EXAMPLE CODE

```
[['peter', 'parker'], [80.0, 70.0, 85.0]],
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

```
def get_stats(class_list):
    new_stats = []
    for elt in class_list:
        new_stats.append([elt[0], elt[1], avg(elt[1])])
    return new_stats

def avg(grades):
    return sum(grades)/len(grades)
```

ERROR IF NO GRADE FOR A STUDENT

- if one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
               [['captain', 'america'], [8.0,10.0,96.0]],
               [['deadpool'], []]]
```

- get `ZeroDivisionError: float division by zero` because try to

```
return sum(grades)/len(grades)
```

length is 0

OPTION 1: FLAG THE ERROR BY PRINTING A MESSAGE

- decide to **notify** that something went wrong with a msg

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

- running on some test data gives

```
no grades data
[['peter', 'parker'], [10.0, 5.0, 85.0], 15.416666666666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.833333333333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], None]]
```

flagged the error

because `avg` did not return anything

OPTION 2: CHANGE THE POLICY

- decide that a student with no grades gets a **zero**

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

- running on some test data gives

```
no grades data
[['peter', 'parker'], [10.0, 5.0, 85.0], 15.416666666666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.833333333333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

still flag the error

now avg returns 0

ASSERTIONS

- want to be sure that **assumptions** on state of computation are as expected
- use an **`assert` statement** to raise an `AssertionError` exception if assumptions not met
- an example of good **defensive programming**

EXAMPLE

```
def avg(grades):
    assert not len(grades) == 0, 'no grades data'
    return sum(grades)/len(grades)
```

function ends immediately if assertion not met

- raises an `AssertionError` if it is given an empty list for grades
- otherwise runs ok

ASSERTIONS AS DEFENSIVE PROGRAMMING

- assertions don't allow a programmer to control response to unexpected conditions
- ensure that **execution halts** whenever an expected condition is not met
- typically used to **check inputs** to functions procedures, but can be used anywhere
- can be used to **check outputs** of a function to avoid propagating bad values
- can make it easier to locate a source of a bug

WHERE TO USE ASSERTIONS?

- goal is to spot bugs as soon as introduced and make clear where they happened
- use as a **supplement** to testing
- raise **exceptions** if users supplies **bad data input**
- use **assertions** to
  - check **types** of arguments or values
  - check that **invariants** on data structures are met
  - check **constraints** on return values
  - check for **violations** of constraints on procedure (e.g. no duplicates in a list)