EXCEPTIONS, ASSERTIONS

EXCEPTIONS AND ASSERTIONS

- what happens when procedure execution hits an **unexpected condition**? something happens I didn't expect -> It's an exception to what we plan for, what we wanted to have happen
- get an **exception**… to what was expected
  - trying to access beyond list limits

    lead typically to a message from Python

    ```
    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

-> able to control: What do I do when I see an exception

```
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")
```

Try says try to execute each of these instructions in turn. But if an exception is raised, stop that processing, jump to the except clause, and execute those statements. And then, carry on outside of the entire loop. If no exception is raised, ignore the except block

- 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

pull things outside of the try body to isolate them as being things I always want to do if it runs correctly.

- `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

(-> "do while" loop)

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

way to control ensuring that the input comes in the right way

only handles ValueErrors! other errors will stop execution

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
```

get a file name by asking for input from the user

try to open the file

Jump out if no file of that name

if that works well, I'm going to open up the file and I'm going to be able to execute out.

process that data, to add new elements in from the data into this variable data so that I can control it*

Close file in either case

* reading in a new line. As long as it isn't just a carriage return, I'm going to take the line, split it by the commas to separate out the pieces to create a list, and then remove using this little thing of taking everything but the last element, removing the trailing carriage return, and add it into the file

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], []])
```

data = []

data = [[student0 name, student0 name, grade0], [student1 name, student1 name, grade1], ...]

as long as I've got some data because I read that in appropriately, which I wanted to check-- then I could loop through it

Handle case of no grade; But assumes two names!

no grade for student

substitute missing data, the grade, with empty list

take out the first two elements-- the student's first and last name-- and the grades and convert that into two lists, a list of the students' name and a list of the actual grades

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[:], []])
```

list evaluated in boolean context, empty list -> False or non-empty list -> True

pull out the name for the student, and I'm going to treat it as everything but the grade-- so everything but the last element. grades be that last element, and I'm going to try and convert it into an integer.

in the case that it can't-- because I don't have something there-- I'm going to catch the value error and simply insert an empty list in that case together with all of the elements of the student

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

let the exception handle the special case

exception: - how to handle exception when I get there- control of deciding when to raise an ex

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

existing error exception (more common) or created by ourselves

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: 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

if I try to index out of range when lists are not of equal length

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

assuming I'm inputting things
that have a list of names and a list of grades.

```
[[['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')
```

None returned if exception raised

- 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
```

explicit return

- running on some test data gives

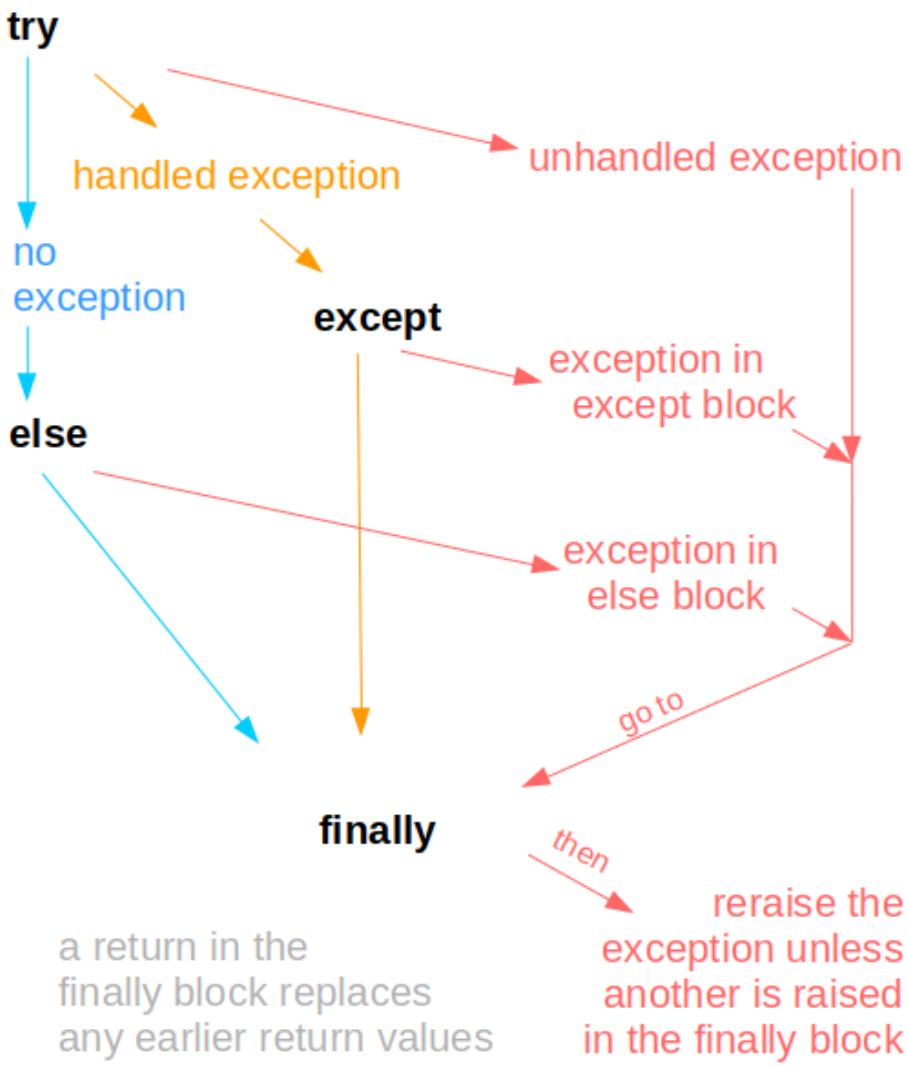
```
no grades data
[['peter', 'parker'], [10.0, 5.0, 85.0], 15.416666666666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.833333333333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

still flag the error

now avg returns 0

Program flow for try (except else finally) statement

docs.python.org/3/reference/compound_stmts.html#the-try-statement
docs.python.org/3/tutorial/errors.html



defensive programming:
think ahead to things that could occur that might not be what I normally expect and think about what I want to do in that case.

-> exception:
- how to handle exception when I get there
- control of deciding when to raise an exception
-> (and how to handle that exception when I get there)

Assertions
- A precondition is something that must be true at the start of a function in order for it to work correctly.
- A postcondition is something that the function guarantees is true when it finishes.
- An invariant is something that is always true at a particular point inside a piece of code.

ASSERTIONS

- want to be sure that **assumptions** on state of computation are as expected

assert that the following should be true, and if it's not, raise an exception

- use an **`assert` statement** to raise an `AssertionError` exception if assumptions not met
- an example of good **defensive programming**

EXAMPLE

expect there to be at least some grades in that list.
And so I can assert that the length of grades will not be 0
if so, assert stement will be skipped

if length of grades will has 0 elements, "not" statement is false.
assert, then, will stop the execution, throw an assert error--
exception with that message printed out

```
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

-> raises an assertion error when I get something I don't want.
Otherwise it runs OK. way of enforcing the expectation
I have for this function. stop immediately if the assertion is not met.
I don't wait til I'm partway through the processing
before I stop the execution of this particular procedure.

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

ASSERTIONS AS DEFENSIVE PROGRAMMING

- 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

bug must have occurred prior to the call to that function.

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)

i.e. list of more than one element