Outline

1 Exception Handling
   - the try-except statement
   - the try-finally statement

2 Python’s Exception Hierarchy
   - exceptions are classes
   - raising exceptions
   - defining exceptions

3 Anytime Algorithms
   - estimating $\pi$ using Monte Carlo
   - handling the KeyboardInterrupt

4 Summary + Assignments

MCS 260 Lecture 28
Introduction to Computer Science
Jan Verschelde, 16 March 2016
what is an exception?

Goal: make code **robust**, capable to handle errors.

An *exception* is an unexpected event that happens during the execution of a program, interrupting the normal flow.

Examples: division by zero in a calculation, wrong user input, or raised by assert statement.

Two phases:

1. the exception is *triggered* (or thrown) by an error, or *raised* explicitly by code in the program;
2. code is executed to *handle* the exception.

Python’s exception mechanism allows programs to handle abnormal situations in a structured way.
We all use elevators:

exception: many people are not afraid to risk body parts impeding closing doors in order to catch a ride

if-else: attendant with manual closing of doors, doubled up: exterior and interior door

For safety, an elevator without monitoring obstacles between closing doors would be unacceptable.

However, exceptions are for exceptional situations.

Frequent use of exceptions is more expensive than the usual if-else construction.
exceptions: defining, raising and handling

1 Exception Handling
   - the try-except statement
   - the try-finally statement

2 Python’s Exception Hierarchy
   - exceptions are classes
   - raising exceptions
   - defining exceptions

3 Anytime Algorithms
   - estimating \( \pi \) using Monte Carlo
   - handling the KeyboardInterrupt

4 Summary + Assignments
The try-except statement

The syntax of the try-except statement is

```
try:
    < code where errors may happen >
except < sequence of exceptions > :
    < code to handle the exception >
```

Three parts in a try-except statement:

1. The code following the `try:` defines the scope of the associated exception handlers.
2. After `except` we specify which exceptions to catch. If the sequence is empty, all exceptions are caught.
3. The code to be executed when the exception occurs, follows the `:` of the `except`.

Multiple except statements may follow one `try` to handle different exceptions differently.
allow for spelling mistakes when prompting for a file

"""
Prompts the user for a file name and displays an error message if the name given by the user does not correspond to a file.
"""

NAME = input('Give the name of a file : ')
try:
    INFILE = open(NAME, 'r')
    print('opened "' + NAME + '" for reading')
    INFILE.close()
except IOError:
    print('file "' + NAME + '" does not exist?')
"""
Prompts the user for a file name and displays an error message if the name given by the user does not correspond to a file. Asks to retry.
"""

while True:
    NAME = input('Give the name of a file : ')
    try:
        INFILE = open(NAME, 'r')
        print('opened "' + NAME + '" for reading')
        INFILE.close()
    except IOError:
        print('file "' + NAME + '" does not exist?')
        RETRY = input('Try again ? (y/n) ')  
        if RETRY != 'y':
            break
exceptions: defining, raising and handling

1 Exception Handling
   - the try-except statement
   - the try-finally statement

2 Python’s Exception Hierarchy
   - exceptions are classes
   - raising exceptions
   - defining exceptions

3 Anytime Algorithms
   - estimating $\pi$ using Monte Carlo
   - handling the KeyboardInterrupt

4 Summary + Assignments
the **try-finally** statement

The syntax of the **try-finally** statement is

```
try:
    < code where errors may happen >
finally:
    < statements before raising exception >
```

Three parts in a **try-finally** statement:

1. The code following the `try:` defines the scope of the associated exception handlers.
2. Statements after `finally:` will be executed even if an exception happens.
3. After executing the statements after `finally:` the exception will be raised.
always confirm user input

Consider the following code:

```python
NAME = input('Give file name : ')  
INFILE = open(NAME, 'r')  
print('opened "' + NAME + '" for reading...')  
```

If the file with given name does not exist, then the last `print` will not be executed.

If we always want to confirm the given name:

```python
NAME = input('Give file name : ')  
try:  
    INFILE = open(NAME, 'r')  
finally:  
    print('opened "' + NAME + '" for reading...')  
```

Even if the file with given name does not exist, `print` will happen before raising `IOError`.
exceptions: defining, raising and handling

1. Exception Handling
   - the try-except statement
   - the try-finally statement

2. Python’s Exception Hierarchy
   - exceptions are classes
   - raising exceptions
   - defining exceptions

3. Anytime Algorithms
   - estimating $\pi$ using Monte Carlo
   - handling the KeyboardInterrupt

4. Summary + Assignments
Object-oriented design is good at expressing hierarchies. Exceptions in Python are classified as below:

Exception is the base class from which the other four exceptions are derived. StandardError and Warning have many subclasses.
the exception hierarchy

```python
>>> from builtins import BaseException
>>> print(BaseException.__doc__)
Common base class for all exceptions
>>> help(BaseException)

Typing help(ValueError) shows:

Help on class ValueError in module builtins:

class ValueError(Exception)
    | Inappropriate argument value (of correct type).
    | Method resolution order:
    |     ValueError
    |     Exception
    |     BaseException
    |     object
```
an example of an exception: `ValueError`

```python
>>> ValueError.__bases__
(<type 'exceptions.StandardError'>,)
>>> issubclass(ValueError, StandardError)
True
```

**When does a `ValueError` happen?**

```python
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```
NameError, TypeError, and IndexError

```python
>>> int(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> 'a'/3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'

>>> 'a'[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```
The following program shows the handling of the keyboard interrupt: cntrl+c.

```
COUNT = 0
print('hold ctrl and c to stop...')
try:
    while True:
        COUNT = COUNT + 1
except KeyboardInterrupt:
    print('counted ', COUNT)
```

Running at the command prompt $:$

$ python cntrlc.py
hold ctrl and c to stop...
^Ccounted  1749022
exceptions: defining, raising and handling

1. Exception Handling
   - the try-except statement
   - the try-finally statement

2. Python’s Exception Hierarchy
   - exceptions are classes
   - raising exceptions
   - defining exceptions

3. Anytime Algorithms
   - estimating $\pi$ using Monte Carlo
   - handling the KeyboardInterrupt

4. Summary + Assignments
raising exceptions: the raise statement

```python
""
Reads numerator and denominator, raises ZeroDivisionError when denominator == 0.
""

try:
    ARAW = input('Give numerator : ')
    BRAW = input('Give denominator : ')
    (A, B) = (int(ARAW), int(BRAW))
    try:
        if B == 0:
            raise ZeroDivisionError
    finally:
        print('read %d/%d' % (A, B))
except ZeroDivisionError:
    print('raised ZeroDivisionError')

Observe the use of finally to show the result, even in case an exception is raised.
```
exceptions: defining, raising and handling

1 Exception Handling
   - the try-except statement
   - the try-finally statement

2 Python's Exception Hierarchy
   - exceptions are classes
   - raising exceptions
   - defining exceptions

3 Anytime Algorithms
   - estimating $\pi$ using Monte Carlo
   - handling the KeyboardInterrupt

4 Summary + Assignments
defining exceptions: exceptions with arguments

We can define our own exceptions. For example, when a file does not exist:

class FileNotThere(IOError):
    
    The exception FileNotThere is raised when a file does not exist. The argument of the exception is the file name.
    
    def __init__(self, name=''):  
    """stores the name of the file""
    IOError.__init__(self)
    self.name = name

FileNotThere inherits from IOError: if not handled explicitly, then it can still be handled via IOError.

The name given by the user is the object data attribute, when handling the exception, we may use the name.
raising exceptions with arguments

An example of raising `FileNotFoundException`:

```python
NAME = input('Give the name of a file : ')
try:
    try:
        INFILE = open(NAME, 'r')
    except IOError:
        raise FileNotThere(NAME)
except FileNotThere as err:
    print('file "' + err.name + '" does not exist')
```

When raising `FileNotFoundException`, we provide the file name. The variable `err` is an instance of `FileNotFoundException`.
exceptions: defining, raising and handling

1. Exception Handling
   - the try-except statement
   - the try-finally statement

2. Python’s Exception Hierarchy
   - exceptions are classes
   - raising exceptions
   - defining exceptions

3. Anytime Algorithms
   - estimating $\pi$ using Monte Carlo
   - handling the KeyboardInterrupt

4. Summary + Assignments
Anytime Algorithms

An anytime algorithm is an algorithm that given some more resources will improve the accuracy of the estimate.

Recall the Monte Carlo method to estimate $\pi$...

Instead of fixing #samples in advance:

1. **use of a while True loop**
2. monitor the progress of the computations via handling of ctrl+c interrupt
3. the handler of `KeyboardInterrupt` shows current approximation and asks user to continue or not
Running `anytimepi.py`

```
$ python anytimepi.py
approximating pi = 3.14159265358979
hold ctrl and c to check...
^C#samples : 1193168, estimate : 3.14061389510949
continue ? (y/n) y
^C#samples : 3733763, estimate : 3.14069746794320
continue ? (y/n) y
^C#samples : 6486987, estimate : 3.14059948015928
continue ? (y/n) y
^C#samples : 11327766, estimate : 3.14173898013077
continue ? (y/n) y
^C#samples : 16384015, estimate : 3.14114434099334
continue ? (y/n) n
```
Monte Carlo Method

"""
Recall the Monte Carlo method to estimate Pi.
"""

from math import pi
from random import uniform as u

print('approximating pi = %.14f' % pi)
(COUNT, TOTAL) = (0, 0)
N = int(input('give number of samples : '))
for i in range(0, N):
    TOTAL += 1
    (X, Y) = (u(0, 1), u(0, 1))
    if X**2 + Y**2 <= 1:
        COUNT += 1
APPROX = (4.0*COUNT)/TOTAL
print('#samples : %d, estimate : %.14f' % (TOTAL, APPROX))
exceptions: defining, raising and handling

1 Exception Handling
- the try-except statement
- the try-finally statement

2 Python’s Exception Hierarchy
- exceptions are classes
- raising exceptions
- defining exceptions

3 Anytime Algorithms
- estimating \( \pi \) using Monte Carlo
- handling the KeyboardInterrupt

4 Summary + Assignments
Pseudo Code

Anytime algorithm with user controlling the stop criterion:

< initialize >
while True:
    try:
        while True:
            < compute estimate >
    except KeyboardInterrupt:
        < show estimate >
        ans = raw_input('continue ? (y/n) ')
        if ans != 'y':
            break

The try–except stays inside the outer while True loop. We leave the outer loop with break.
an anytime Python script to estimate $\pi$

```
from math import pi
from random import uniform as u

print('approximating pi = %.14f' % pi)
print('hold ctrl and c to check...')
(COUNT, TOTAL) = (0, 0)
while True:
    try:
        while True:
            TOTAL += 1
            (X, Y) = (u(0, 1), u(0, 1))
            if X**2 + Y**2 <= 1:
                COUNT += 1
    except KeyboardInterrupt:
        APPROX = (4.0*COUNT)/TOTAL
        print('#samples : %d, estimate : %.14f' % (TOTAL, APPROX))
        ANS = input('continue ? (y/n) ')
        if ANS != 'y':
            break
```
Assignments:

1. Write a function that prompts the user for an age. Raise `Exception` when the age is negative.

2. Define an exception `InvalidAge` that has the age as object data attribute.

3. Give an illustration of how to raise the exception defined in previous exercise.