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
Exception Handling

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.
Exception Handling

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.
Exception Handling

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.
Exception Handling

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.
Exception Handling

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.
Exception Handling

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.
Closing Elevator Doors

a real world example

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.
Closing Elevator Doors

a real world example

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.
Closing Elevator Doors
a real world example

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.
Closing Elevator Doors

a real world example

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.
Closing Elevator Doors  

a real world example

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 >
extcept < 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.
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.
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.
The **try-except** Statement

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

```python
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.
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

```python
# L-28 MCS 260 Wed 29 Oct 2008 : readfilename.py

# 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 = raw_input('Give the name of a file : '
try:
    file = open(name,'r')
    print 'opened ' + name + ' for reading'
    file.close()
except IOError:
    print 'file ' + name + ' does not exist?'
```
Allow for Spelling Mistakes
when prompting for a file

```python
# L-28 MCS 260 Wed 29 Oct 2008 : readfilename.py

# 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 = raw_input('Give the name of a file : ')
try:
    file = open(name,'r')
    print 'opened ' + name + ' for reading'
    file.close()
except IOError:
    print 'file ' + name + ' does not exist?'
```
Allow for Spelling Mistakes
when prompting for a file

# L-28 MCS 260 Wed 29 Oct 2008 : readfilename.py

# 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 = raw_input('Give the name of a file : ')
try:
    file = open(name,'r')
    print 'opened ' + name + ' for reading'
    file.close()
except IOError:
    print 'file ' + name + ' does not exist?'
Exception Handling
the try-except statement
the try-finally statement
Python's Exception Hierarchy
exceptions are classes raising exceptions defining exceptions
Anytime Algorithms estimating π using Monte Carlo handling the KeyboardInterrupt
Summary + Assignments

Keep on Trying
use in while True

# L-28 MCS 260 Wed 29 Oct 2008 : readfilename2.py

# 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 = raw_input('Give the name of a file : ')
    try:
        file = open(name, 'r')
        print 'opened ' + name + ' for reading'
        file.close()
        break
    except IOError:
        print 'file ' + name + ' does not exist?'
        retry = raw_input('Try again ? (y/n) ')
        if retry != 'y': break
Keep on Trying
use in while True

```python
# L-28 MCS 260 Wed 29 Oct 2008 : readfilename2.py

# 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 = raw_input('Give the name of a file : ')
    try:
        file = open(name,'r')
        print 'opened ' + name + ' for reading'
        file.close()
    break
    except IOError:
        print 'file ' + name + ' does not exist?'
        retry = raw_input('Try again ? (y/n) ')
        if retry != 'y': break
```
Exception Handling

the try-except statement

the try-finally statement

Python's Exception Hierarchy

exceptions are classes
raising exceptions
defining exceptions

Anytime Algorithms

estimating $\pi$ using Monte Carlo
handling the KeyboardInterrupt

Summary + Assignments

# L-28 MCS 260 Wed 29 Oct 2008 : readfilename2.py

# 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 = raw_input('Give the name of a file : ')
    try:
        file = open(name,'r')
        print 'opened ' + name + ' for reading'
        file.close()
        break
    except IOError:
        print 'file ' + name + ' does not exist?'
        retry = raw_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.
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.
The try-finally Statement

The syntax of the try-finally statement is

\begin{verbatim}
try:
    < code where errors may happen >
finally:
    < statements before raising exception >
\end{verbatim}

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.
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.
Consider the following code:

```python
name = raw_input('Give file name : ')  
file = 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 = raw_input('Give file name : ')  
try: 
    file = 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`.

**Summary + Assignments**

Always Confirm User Input

*an example of try-finally*
Always Confirm User Input
an example of try-finally

Consider the following code:

```python
ame = raw_input('Give file name : ')
file = 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 = raw_input('Give file name : ')
try:
    file = 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.
Consider the following code:

```python
name = raw_input('Give file name : ')
file = 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 = raw_input('Give file name : ')
try:
    file = 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.
Always Confirm User Input
an example of try-finally

Consider the following code:

```python	name = raw_input('Give file name : ')

try:
    file = 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 = raw_input('Give file name : ')

try:
    file = 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.
Always Confirm User Input
an example of try-finally

Consider the following code:

```python
name = raw_input('Give file name : ')
file = 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 = raw_input('Give file name : ')
try:
    file = 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.
Exception Handling
the try-except statement
the try-finally statement

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

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

Summary + Assignments
Object-oriented design is good at expressing hierarchies.

Exceptions in Python are classified as below:

```
Exception
  ├── GeneratorExit
  │    ├── StandardError
  │    └── Warning
  └── StopIteration
```

Exception is the base class from which the other four exceptions are derived.

StandardError and Warning have many subclasses.
Object-oriented design is good at expressing hierarchies. Exceptions in Python are classified as below:

- Exception
- GeneratorExit
- StandardError
- StopIteration
- Warning

Exception is the base class from which the other four exceptions are derived. StandardError and Warning have many subclasses.
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.
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.
Object-oriented design is good at expressing hierarchies. Exceptions in Python are classified as below:

```
Exception

GeneratorExit

StandardError

StopIteration

Warning
```

Exception is the base class from which the other four exceptions are derived. StandardError and Warning have many subclasses.
Python’s Exception Hierarchy

exceptions are classes

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

```
Exception

GeneratorExit

StandardError

Warning

StopIteration
```

Exception is the base class from which the other four exceptions are derived. StandardError and Warning have many subclasses.
Object-oriented design is good at expressing hierarchies. Exceptions in Python are classified as below:

```
Exception
    ├── GeneratorExit
    │      └── StandardError
    └── Warning
        └── StopIteration
```

Exception is the base class from which the other four exceptions are derived.

StandardError and Warning have many subclasses.
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 Complete Exception Hierarchy on any Python installation

>>> import exceptions

>>> print exceptions.__doc__

Python’s standard exception class hierarchy.

Exceptions found here are defined both in the exceptions module and the built-in namespace. It is recommended that user-defined exceptions inherit from Exception. See the documentation for the exception inheritance hierarchy.

>>> help(exceptions)

The last command shows the complete hierarchy.
The Complete Exception Hierarchy

on any Python installation

>>> import exceptions
>>> print exceptions.__doc__
Python’s standard exception class hierarchy.

Exceptions found here are defined both in the exceptions module and the built-in namespace. It is recommended that user-defined exceptions inherit from Exception. See the documentation for the exception inheritance hierarchy.

>>> help(exceptions)

The last command shows the complete hierarchy.
The Complete Exception Hierarchy
on any Python installation

>>> import exceptions
>>> print exceptions.__doc__

Python’s standard exception class hierarchy.

Exceptions found here are defined both in the exceptions module and the built-in namespace. It is recommended that user-defined exceptions inherit from Exception. See the documentation for the exception inheritance hierarchy.

>>> help(exceptions)

The last command shows the complete hierarchy.
# L-28 MCS 260 Wed 29 Oct 2008 : cntrlc.py

# The following program shows the handling of the keyboard interrupt: cntrl+c.

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

$ python cntrlc.py
hold ctrl and c to stop...
^Ccounted  1749022
# L-28 MCS 260 Wed 29 Oct 2008 : cntrlc.py

# The following program shows the handling # of the keyboard interrupt: cntrl+c.

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

$ python cntrlc.py
hold ctrl and c to stop...
^Ccounted  1749022
# L-28 MCS 260 Wed 29 Oct 2008 : cntrlc.py

# The following program shows the handling
# of the keyboard interrupt: cntrl+c.

cnt = 0
print 'hold ctrl and c to stop…'
try:
    while True:
        cnt = cnt + 1
except KeyboardInterrupt:
    print 'counted ', cnt

$ python cntrlc.py
hold ctrl and c to stop...
^Ccounted  1749022
# L-28 MCS 260 Wed 29 Oct 2008 : cntrlc.py

# The following program shows the handling of the keyboard interrupt: cntrl+c.

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

$ python cntrlc.py
hold ctrl and c to stop...  
^Ccounted  1749022
The following program shows the handling of the keyboard interrupt: cntrl+c.

cnt = 0
print 'hold ctrl and c to stop...'
try:
    while True:
        cnt = cnt + 1
ect except KeyboardInterrupt:
    print 'counted ', cnt

$ 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
# L-28 MCS 260 Wed 29 Oct 2008 : ratraise.py

# Reads numerator and denominator, raises
# ZeroDivisionError when denominator == 0.

try:
a = input('Give numerator : ')
b = input('Give denominator : ')
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.
Raising Exceptions

the raise statement

```
# L-28 MCS 260 Wed 29 Oct 2008 : ratraise.py

# Reads numerator and denominator, raises
# ZeroDivisionError when denominator == 0.

try:
    a = input('Give numerator : ')
    b = input('Give denominator : ')
    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.
Raising Exceptions
the raise statement

```python
# L-28 MCS 260 Wed 29 Oct 2008 : ratraise.py

# Reads numerator and denominator, raises
# ZeroDivisionError when denominator == 0.

try:
    a = input('Give numerator : ')
    b = input('Give denominator : ')
    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
We can define our own exceptions. For example, when a file does not exist:

```python
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=''):  
        self.name = name
```

FileNotThere inherits from IOError: if not handled explicitly, 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.
We can define our own exceptions. For example, when a file does not exist:

```python
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=''):  
        self.name = name

FileNotThere inherits from IOError: if not handled explicitly, 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.
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=''):  
        self.name = name

FileNotThere inherits from IOError: if not handled explicitly, 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.
We can define our own exceptions. For example, when a file does not exist:

```python
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=''):  
        self.name = name

FileNotThere inherits from IOError: if not handled explicitly, 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 `FileNotThere`:

```python
name = raw_input('Give the name of a file : ')
try:
    try:
        file = open(name,'r')
    except IOError:
        raise FileNotThere(name)
except FileNotThere,e:
    print 'file ' + e.name + ' does not exist'
```

When raising `FileNotThere`, we provide the file name. The variable `e` is an instance of `FileNotThere`.
Raising Exceptions with Arguments

An example of raising `FileNotFoundException`:

```python
name = raw_input('Give the name of a file : ')  
try:
    try:
        file = open(name,'r')
    except IOError:
        raise FileNotThere(name)
except FileNotThere,e:
    print 'file ' + e.name + ' does not exist'
```

When raising `FileNotFoundException`, we provide the file name. The variable `e` is an instance of `FileNotFoundException`.
Raising Exceptions with Arguments

An example of raising `FileNotFoundException`:

```python
name = raw_input('Give the name of a file : ')  
try:                                               
    try:                                           
        file = open(name,'r')                      
    except IOError:                                
        raise FileNotThere(name)                   
except FileNotThere,e:                            
    print 'file ' + e.name + ' does not exist'
```

When raising `FileNotFoundException`, we provide the file name. The variable `e` is an instance of `FileNotFoundException`. 
Raising Exceptions with Arguments

An example of raising `FileNotFoundException`:

```python
name = raw_input('Give the name of a file: ')  
try:
    try:
        file = open(name, 'r')
    except IOError:
        raise FileNotFoundException(name)
except FileNotFoundException, e:
    print 'file ' + e.name + ' does not exist'
```

When raising `FileNotFoundException`, we provide the file name. The variable `e` 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 `cntrl+c` interrupt
3. the handler of `KeyboardInterrupt` shows current approximation and asks user to continue or not
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 `cntrl+c` interrupt
3. the handler of `KeyboardInterrupt` shows current approximation and asks user to continue or not
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 cntrl+c interrupt
3. the handler of KeyboardInterrupt shows current approximation and asks user to continue or not
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 cntrl+c interrupt**
3. **the handler of KeyboardInterrupt shows current approximation and asks user to continue or not**
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 `cntrl+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
```
$ 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
$ 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
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
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
```
import math
from random import uniform as u

print 'approximating pi = %.14f' % math.pi
(count, total) = (0, 0)
n = 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
p = (4.0*count)/total
print '#samples : %d, estimate : %.14f' % (total, p)
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 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 outer while True. We leave the outer loop with break.
The Code

import math
from random import uniform as u

print 'approximating pi = %.14f' % math.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:
        p = (4.0*count)/total
        print '#samples : %d, estimate : %.14f' 
        % (total,p)
        ans = raw_input('continue ? (y/n) ')
        if ans != 'y': break
import math
from random import uniform as u

print 'approximating pi = %.14f' % math.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:
        p = (4.0*count)/total
        print '#samples : %d, estimate : %.14f' % (total, p)
        ans = raw_input('continue ? (y/n) ')
        if ans != 'y': break
import math
from random import uniform as u

print 'approximating pi = %.14f' % math.pi
print 'hold ctrl and c to check...'
(count, total) = (0, 0)
while True:
    try:
        total += 1
        (x, y) = (u(0, 1), u(0, 1))
        if x**2 + y**2 <= 1: count += 1
    except KeyboardInterrupt:
        p = (4.0*count)/total
        print '#samples : %d, estimate : %.14f' % (total, p)
        ans = raw_input('continue ? (y/n) ')
        if ans != 'y': break
Summary + Assignments

See chapter 8 in *Python Power!*

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.