12:Handling errors

Validating user input

- Validating user input is extremely important in programs to
  - Prevent errors when unacceptable input is received,and
  - Notify user to give valid input values

Non valid input values

- There are several examples of erroneous input values, for example
  - Out of range values (beyond sequence indeces, larger than the object type supports, too precise to fit into object type etc.)
  - Wrong type values (expecting integer / found a string, binary file instead of text file, wrong format in text file or sequence...)
  - Empty values (non-existing or empty file, no query value, parameter missing...)

Validating input

- We already know some methods for validating input:
  - Checking the boundaries can be done by conditional statements
  - Checking the type can be done with `isinstance` function.

Example

- Only process integer values between 1 and 10:

```
n = input("Give a value: ")
if isinstance(n, int) and 1 <= n <= 10:
    # do something
else:
   print "Not a valid value"
```

However…

- The example throws an error, if the user enters a string instead of a number.
- To detect mistakes like these, we need a mechanism for catching errors.

Exceptions in Python

- 'Expected' errors can be caught with a try – except statements.
- The syntax:

```
try:
   # something that may fail
except <Error>:
   # there was an error: do something
```

Try – except

- If an error occurs in the **try** block, the block is terminated. Python then checks whether there is an **except** block for that *error type*.
- If one is found, the execution continues inside the except block. After that, the execution continues normally after that block.
- If one is not found, the exception is *passed on to outer try statements.* If none is found, the program terminates.

Example:

- Catch errors in input:

```
try:
   n = input("give a number: ")
   print "Number * 2 = ", n * 2
except:
   print "Something went wrong!"
```

Catching all errors

- Catching all errors with except statement without specifying an error is a *dangerous* programming convention
- It catches the *errors made by programmer* as well as the user made mistakes!

The types of errors

- There are several error types in Python, for example
  - **TypeError**
  - **ValueError**
  - **ZeroDivisionError**
  - **AttributeError**
  - **NameError**
  - **IOError**
- Complete listing can be found at http://docs.python.org/library/exceptions.html

TypeError

- Raised when an operation or function is applied to an object of inappropriate type.
- For example:

```
print "a" + 3

a = "123"
a.find(1)
```

ValueError

- Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
- Example:

```
print int("Hello!")
```

ZeroDivisionError

- Raised, when an attempt to divide with a zero is made.
- Example:

```
print 3 / 0
```

AttributeError

- Raised when an attribute reference or assignment fails.
- For example:

```
a = 3
a.sort() # int object has no attribute 'sort'
```

NameError

- Raised when a local or global name is not found.
- For example:

```
getValues() # if no such method

print value # if value not initialized
```

IOError

- Raised, when a file read or write error happens.
- Example:

```
f = open("the_file.txt", "r") # if no such file

f = open("newfile.dat", "w")
f.readlines() # file not opened for reading
```

User made vs. programmer made errors

- The basic idea is, that the exceptions are to be used to catch *user made errors*.
- *Programmer made errors* should be catched by compiler errors and testing!

Catching errors

- Using the except statement without specifying the error catches *all errors,* including those made by the programmer!
- Hence, you should always specify the error type(s) when using try – except.

Catching errors by type

- A specific error can be catched by using the error name:

```
try:
   f = open("myfile.txt", "r")
except IOError:
   print "The file can not be opened."
   print "Please check, that the file exists."
```

- ...catches errors in disk operations, but **not** programmer made errors.

Catching errors by type (cont.)

- Consider the other example:

```
try:
   f = open("myfile.txt", "s")
except:
   print "The file can not be opened."
   print "Please check, that the file exists!"
```

- …displays an error message, because the programmer made an error (there is no such file mode as "s"). User has no way to correct the behaviour.

Catching errors by type (cont.)

- It is possible to specify several error types in a tuple:

```
try:
   # do something here…
except (TypeError, IOError, ValueError):
   # catches three types of errors
```

Catching errors by type

- It is also possible to have several except blocks with one try block:

```
try:
   # something done here
except ValueError:
   # handle ValueError
except TypeError:
   # Handle TypeError
except IOError:
   # Handle IOError
```

Passing errors to callers

- If an error is not caught, it is passed up to caller of the routine.
- Example:

```
try:
   myMethod() # calls myMethod..
except ValueError:
   # if a ValueError was raised & not caught
   # in myMethod, it is passed here.
```

Raising errors

- It is also possible to raise an error manually in a program.
- An error is raised with a **raise** keyword.
- Syntax:

```
raise Error(error message)
```

- When an error is raised, the execution of the function is immediately terminated.

Raising errors (cont.)

- Example: raise a TypeError, if the function expecting an integer had a different type argument:

```
def factorial(value):
""" returns factorial value! """
   if not isinstance(value, int):
      raise TypeError("Integer expected!")
   # calculate and return factorial…
```

Raising errors (cont.)

- Example 2: Raise a ValueError, if the argument provided is negative:

```
def factorial(value):
""" returns factorial value! """
   if not isinstance(value, int):
      raise TypeError("Integer expected!")
   if value < 0:
      raise ValueError("value must be >= 0!")
   # calculate and return factorial…
```

Finally…

Some topics that didn't fit in other categories

The pass keyword

- Python has a keyword **pass**, which can be used to do *nothing*. It is used, when a statement is required syntactically, but you don't want to actually execute anything.
- For example, you can use it as a place-holder, when you want to implement a function later:

```
def function():
   pass # this is defined later…
```

Lambda functions

- The **lambda** keyword can be used to define mini-functions, i.e. functions that return the result of a single expression.
- The syntax:

```
lambda <variable(s)> : <operation
          involving variable(s)>
```

Lambda functions (cont.)

- Since the lambda returns a *new function*, it should be either passed as an argument, or stored in the variable to use it later:

```
adder = lambda x : x + 1
print adder(1) # outputs 2
print adder(23) # outputs 24
```

Lambda functions (cont.)

- Example: a mini-function, that creates an integer list by converting all items in an existing list into integers:

```
int_list = lambda x : [int(y) for y in x]
s = ["1","2","4","8","16"]
s2 = int_list(s)
print s2
(1, 2, 4, 8, 16)
```

Lambda functions (cont.)

- Lambda also accepts more than one argument:
- Consider these examples:

```
greater = lambda x,y : x > y
print greater(2,1) # output True
print greater(1,2) # output False

multi = lambda a,b,c : a * b * c
print multi(1,2,3) # output 6
print multi(2,2,2) # output 8
```

Functions as arguments

- It is possible to pass functions as arguments in Python. Consider this example:

```
def operate(val1, val2, operation):
   return operation(val1,val2)

print operate(3, 2, lambda x,y : x + y) # 5
print operate(3, 2, lambda x,y: x * y) # 6
print operate(11, int, isinstance) # True
```

The dir function

- The built-in dir function can be used to list all operations in a module.
- Example:

```
import math
dir(math) # lists all operations in math
```

String formatting

- The format() method can be used to format strings for output. Example:

```
a = 10
b = 20
print "{0} + {1} = {2}".format(a, b, a + b)
```

- See http://docs.python.org/library/string.html#formatstrings for more examples.

Finally, things we didn't discuss (but which Python can do…)

- Classes, Object oriented programming
- Graphical user interfaces
- Advanced IO (mouse, keyboard, USB, etc…)
- Graphics
- Networking