Exceptions

Creating errors

Creating an error is called “raising an exception”. You may raise an exception like this:

```python
raise Exception("Something went wrong")
```

Typical exceptions are

- `TypeError`
- `ValueError`
- `IndexError`

You already know `SyntaxError` and `ZeroDivisionError`.
Review the alarms

Reviewing the errors

You may review the errors using `try` and `except`:

```python
try:
    <some code that might raise an exception>
except ValueError:
    print "Oops, a ValueError occurred"
except TypeError:
    print "We got a TypeError but not a ValueError"
except Exception:
    print "Some other kind of exception occurred."
```
Review the alarms

Reviewing the errors

You may review the errors using try and except:

```python
try:
    <some code that might raise an exception>
except ValueError:
    print "Oops, a ValueError occurred"
except TypeError:
    print "We got a TypeError but not a ValueError"
except Exception:
    print "Some other kind of exception occurred."
```

**Flow Control**

An exception stops the flow and looks for the closest enclosing try block. If it is not caught it continues searching for the next try block.
Error messages

*Golden rule*

Never print error messages, *raise an exception instead*
Error messages

**Golden rule**

Never print error messages, *raise an exception instead*

Don’t do this:

```python
def fixpoint_iter(f, x0, maxit = 100, tol = 1e-6):
    x = copy.copy(x0)
    for i in range(maxit):
        fx = f(x)
        if abs(fx-x) < tol:
            break
        x = fx
    else:
        print('The iteration did not converge in {0} iterations'.format(maxit))
    return x, i
```
Error messages

Golden rule
Never print error messages, *raise an exception instead*

Do this:

```python
def fixpoint_iter(f, x0, maxit = 100, tol = 1e-6):
    x = copy.copy(x0)
    for i in range(maxit):
        fx = f(x)
        if abs(fx-x) < tol:
            break
        x = fx
    else:
        raise Exception('The iteration did not converge in {0}
    return x, i
```
Catching the error

With the last construction we can do things like this:

```python
for it in [10, 100, 1000, 10000]:
    try:
        x, numit = fixpoint_iter(f, x0, maxit = it)
    except Exception:
        print('{} iterations were not enough.'.format(it))
    else:
        print('{} iterations were enough!'.format(it))
        print('Converged to {} in {} iterations'.format(x, numit))
        break
```
File I/O

File I/O (in- and output) is essential when

- working with measured or scanned data
- interacting with other programs
- saving information for comparisons or other postprocessing needs
- ….
File objects

A file is a Python object with associated methods:

```python
# Creating a read-only file object
myfile = open('measurement.dat', 'r')
```

The whole file can be read and stored in a string by

```python
s = myfile.read()
```

You can also read it one line at a time by

```python
myfile.readline()
```

or like

```python
for line in myfile.readlines()
    print line
```
Files as generators

A file object is a *generator*. We will talk more about generators later.

A generator is like a list, except the values need not exist until asked for.

A main feature of generators is that they are *disposable*. When you read a line from a file, it is removed from the file object (not from the file itself). The following code will print three different things:

```python
print(myfile.readline())  # Line 1
print(myfile.readline())  # Line 2
print(myfile.readline())  # Line 3
```
File close method

A file has to be closed before it can be reread.

```python
myfile.close() # closes the file object
```

It is automatically closed when

- the program ends
- the enclosing program unit (e.g. function) is left.

Before a file is closed, you won’t see any changes in it by an external editor!
The \texttt{with} statement

If you forget to close a file, problems can occur. Also, an error might prevent you from closing the file. Consider

```python
myfile = \texttt{open}(\texttt{name}, \texttt{w})
myfile.write('some data')
a = 1/0
myfile.write('other data')
myfile.close()
```

The \texttt{with} statement helps with this:

```python
\texttt{with open(name, w) as myfile:}
    \texttt{myfile.write('some data')}
a = 1/0
    \texttt{myfile.write('other data')}
```

With this construction, the file is always closed even if an exception occurs. It is shorthand for a clever \texttt{try}-\texttt{except} block.
**File Modes (read, write, etc.)**

```python
file1 = open('file1.dat', 'r')  # read only
file2 = open('file2.dat', 'r+')  # read/write
file3 = open('file3.dat', 'a')  # append
file4 = open('file4.dat', 'w')  # (over-)write
```

The modes 'r', 'r+', 'a' require that the file exists.

**File append example**

```python
file3 = open('file3.dat', 'a')
file3.write('something new
')  # Note the \n'
```
Saving NumPy arrays

The read and write methods convert data to strings. Complex data types (like arrays) cannot be written this way. NumPy provides its own methods for storing arrays.

```python
a = array([10, 20, 30])
with open('outfile.txt', 'w') as myfile
    numpy.savetxt(myfile, a)  # Saves a in outfile.txt

with open('outfile', 'w') as myfile
    numpy.save(myfile, a)  # Saves a in outfile (binary)
```

You can also just give the name of the file as a string:

```python
numpy.savetxt('outfile.txt', a)  # Saves a in outfile.txt
numpy.save('outfile', a)  # Saves a in outfile.npy (binary)
```

By using `numpy.savez` one can save several arrays in one file.
**Reading NumPy arrays**

Reading arrays is done similarly

```python
with open('infile.txt', 'r') as myfile
    a = numpy.loadtxt(myfile)

with open('infile.npy', 'r') as myfile
    numpy.load(myfile, a)
```

or

```python
a = numpy.loadtxt('infile.txt')
a = numpy.load('infile.npy')
```

If the data is separated by something other than spaces you should use the `delimiter` keyword

```python
a = numpy.loadtxt('infile.txt', delimiter = ';')
# Turns '1; 2; 3;' into array([1,2,3])
```

There are several more options like this, see the documentation.
Module: pickle

If we have other objects than arrays, which are difficult to convert to strings, we can use the pickle module.

Pickle dump examples

```python
import pickle
with open('file.dat', 'w') as myfile:
a = rand(200)
b = 'a string'
pickle.dump(a, myfile)  # first call: first object
pickle.dump(b, myfile)  # second call: second object
```

You can pickle (almost) any Python object. Even code, e.g. functions.
Module: pickle

Pickle load examples

```python
import pickle
myfile = open('file.dat', 'r')
numbers = pickle.load(myfile)  # restores the array
text = pickle.load(myfile)    # restores the text
```
Module: shelve

When pickling data you have to load them in the correct order. An extension to the pickle module is the shelve module. This basically works like a dictionary stored in a file.

```python
import shelve
d = shelve.open(filename)

d['N'] = 5
d['mylist'] = ['a', 'b', 'c']
d['myarray'] = arange(d['N'])

d.items()  # [('N', 10), ('mylist', ['a', 'b', 'c']),
# ('myarray', array([0, 1, 2, 3, 4]))]

# NOTE: modifying items does not modify the file
# by default but it fails *silently*
d['mylist'].append('d')
d['mylist']  # ['a', 'b', 'c']

d.close()  # Writes to file
```