Introduction to Programming using PYTHON

Session 5

N. D. Mendes
ndm@algos.inesc-id.pt

INESC-ID

October 19, 2006

Part I

Recursive Functions

Recursive Functions

Exercise

Define a iterative and a recursive version of a function implementing the mathematical `factorial` function (assuming you always get a non-negative number).

Recursive Functions

Exercise

Define a iterative and a recursive version of a function implementing the mathematical `factorial` function (assuming you always get a non-negative number).

Iterative version

```
def factorial(n):
  res = 1
  for i in range(2,n):
   res *= i
  return res
```

Recursive Functions

Exercise

Define a iterative and a recursive version of a function implementing the mathematical `factorial` function (assuming you always get a non-negative number).

Iterative version

```
def factorial(n):
  res = 1
  for i in range(2,n):
   res *= i
  return res
```

Recursive version

```
def factorial(n):
  if n <= 1:
   return 1
  return n * factorial(n-1)
```

Part II

Exceptions

Exceptions

- Exceptions are a useful mechanism to change the normal flow of a program in exceptional circumstances
- Exceptions come in different types. There are built-in types (e.g. `ValueError`, `ZeroDivisionError`) and user-defined types

Consider the following example:

```
while True:
  try:
   x = int(raw_input("Give me a number:  "))
   break
  except ValueError:
   print "Invalid number, try again.."
```

You can catch a number of exception in a single except clause

```
...
except (RunTimeError, TypeError, NameError):
 pass
```

The last `except` clause can be a wildcard:

```
...
except:
 print "This catches any exception"
```

The last `except` clause can also be used to re-raise the exception and have someone else deal with it.

```
import sys

try:
  f = open("myfile.txt")
  s = f.readline()
  i = int(s.strip())
except IOError, (errno, strerror):
  print "I/O error(%s):  %s" % (errno, strerror)
except ValueError:
  print "Could not convert data to an integer."
except:
  print "Unexpected error:", sys.exc_info()[0]
  raise
```

Exceptions
Handling exceptions

The `else` clause can also be useful because it allows you to isolate the code to be protected inside the `try ... except` statement and prevent you from inadvertently catching an exception raised by additional code.

```
for arg in sys.argv[1:]:
  try:
   f = open(arg, "r")
  except IOError:
   print "cannot open", arg
  else:
   print arg, "has", len(f.readlines()), "lines"
   f.close()
```

```
try:
   raise Exception("spam", "eggs")
except Exception, inst:
    print type(inst)   # the exception instance
    print inst.args    # arguments stored in .args
    print inst         # __str__ lets args be printed directly
    x, y = inst        #__getitem__ lets args be unpacked directly
    print "x =", x
    print "y =", y
```

The mechanism of exceptions is useful because it allows you to deal with erros in the most appropriate place. The `try...except` statement will catch exceptions raised by any code invoked inside the `try` block.

```
def this_fails():
  x = 1/0

try:
  this_fails()
except ZeroDivisionError, detail:
  print "Handling run-time error:", detail
```

Raising Exceptions

```
raise Exception
raise Exception, "Optional Message"
raise Exception("Optional Message")
raise Exception("Optional Message", "Yet another
argument")
```

```
try:
  raise NameError, "HiThere"
except NameError:
  print "An exception flew by!"
 raise
```

Raising Exceptions

```
raise Exception
raise Exception, "Optional Message"
raise Exception("Optional Message")
raise Exception("Optional Message", "Yet another
argument")
```

```
try:
  raise NameError, "HiThere"
except NameError:
  print "An exception flew by!"
 raise
```

User-defined exceptions will be covered in the next sessions

For the next session

- From the manual
  - Read chapter 17
- Continue working of Series 1
- A Series 2 will be given in the next session