User input/output, error handling and modules Part II

Foundation of programming (CK0030)

Francesco Corona

FdP

- ☺ Intro to variables, objects, modules, and text formatting
- ☺ Programming with WHILE- and FOR-loops, and lists
- ☺ Functions and IF-ELSE tests

- ☺ Data reading and writing
- ☹ **Error handling**
- ☹ Making modules

- ☹ Arrays and array computing
- ☹ Plotting curves and surfaces

Handling errors
User input, error handling and modules

Handling errors

```
1 import sys
2 C = float(sys.argv[1])
3 F = 9.0*C/5 + 32
4 print F
```

Suppose we forget to provide a command-line argument to `c2f_cml.py`

```
1 Terminal > c2f_cml.py
2  Traceback (most recent call last):
3   File "c2f_cml.py", line 2, in ?
4    C = float(sys.argv[1])
5  IndexError: list index out of range
```

Python aborts the program and shows an error message containing

- The line where the error occurred
- The type of error (`IndexError`)
- An explanation of what the error is

Handling errors (cont.)

```
1 import sys
2 C = float(sys.argv[1])
3 F = 9.0*C/5 + 32
4 print F
```

From this info we deduce that index 1 of sys.argv is out of range

```
1 Terminal > c2f_cml.py
2  Traceback (most recent call last):
3   File "c2f_cml.py", line 2, in ?
4    C = float(sys.argv[1])
5  IndexError: list index out of range
```

As there are no command-line arguments, sys.argv has one element

- The program name itself
- The only valid index is 0

Handling errors (cont.)

For an experienced coder this error message
is clear enough to indicate what is wrong

For others it would be more helpful if wrong usage could be detected by the program and a description of correct operation could be printed

The question is how to detect the error inside the program

Handling errors (cont.)

The problem in the execution is that `sys.argv` does not contain two elements (program name, as always, and a command-line argument)

We can test on the length of `sys.argv` to detect wrong usage

- if `len(sys.argv)` is less than `2`, info on `C` is missing

Example

A new version of the program, `c2f_cml_if.py`, starts with an `IF test`

```
1 if len(sys.argv) < 2:
2  print 'You failed to provide Celsius degrees'\
3        'as input on the command line!'
4  sys.exit(1)                              # abort because of error
5
6 F = 9.0*C/5 + 32
7 print '%gC is %.1fF' % (C, F)
```

Handling errors (cont.)

`sys.exit` is used to abort execution and show an error code

- Any non-zero argument means the program aborted due to error
- The precise value of the argument does not matter
- Here, we choose it to be `1`

If no errors are found, we can still use `sys.exit(0)` to abort it

Handling errors (cont.)

The modern way of handling errors in a code is to try to execute some statements, and if something goes wrong, the program can detect this and jumps to a set of statements that handle the erroneous situation

- The **Try-Except construction**

Definition

The relevant `TRY-EXCEPT construction` reads

```
try:
 <try-block statements>
except:
 <except-block statements>
```

If the **attempt** goes wrong in the `TRY block`, an **exception** is raised

- The execution jumps directly to the `EXCEPT block`

Exception handling
Handling errors

Exception handling

To clarify the idea of exception handling, consider a TRY-EXCEPT block to handle a problem with the Celsius-Fahrenheit conversion program

- A wrong call lacking a command-line argument

Example

```
1  import sys
2  try:
3   C = float(sys.argv[1])
4  except:
5   print 'You failed to provide Celsius degrees'\
6         'as input on the command line!'
7   sys.exit(1) # abort
8
9  F = 9.0*C/5 + 32
10 print '%gC is %.1fF' % (C, F)
```

If an argument is missing, indexing `sys.argv[1]` raises an exception

- The program jumps directly to the EXCEPT block
- The `float` is not called, and `C` is not initialised

In the EXCEPT block, info about exception is given and code executed

- We print a message and abort the program

Exception handling (cont.)

If the command-line argument is provided, the `TRY block` is executed successfully, the program neglects the `EXCEPT block` and continues

```
1 Terminal > c2f_cml_except1.py
2  You failed to provide Celsius degrees
3  as input on the command line!
4
5 Terminal > c2f_cml_except1.py 21
6  21C is 69.8F
```

Exception handling (cont.)

Consider the assignment

```
1 C = float(sys.argv[1])
```

There are two typical errors associated with this statement

- `sys.argv[1]` is illegal indexing when no CL arguments are given
- String `sys.argv[1]` can not be converted into a `float object`

Python detects both these errors and raises an exception

- An `IndexError` exception in the first case
- A `ValueError` exception in the second case

Exception handling (cont.)

In the program, we jump to the EXCEPT block and issue the same exact message, regardless of what went wrong in the TRY block

```
1  import sys
2  try:
3   C = float(sys.argv[1])
4  except:
5   print 'You failed to provide Celsius degrees'\
6         'as input on the command line!'
7   sys.exit(1) # abort
8
9  F = 9.0*C/5 + 32
10 print '%gC is %.1fF' % (C, F)
```

Exception handling (cont.)

When we provide a command-line argument, but write it wrongly, the program jumps to the `EXCEPT block` and prints a misleading message

```
1 Terminal > c2f_cml_except1.py 21C
2  You failed to provide Celsius degrees
3  as input on the command line!
```

Exception handling (cont.)

A solution to this could be to branch into different `EXCEPT blocks` depending on what type of exception was raised in the `TRY block`

```
1  import sys
2
3  try:
4   C = float(sys.argv[1])
5  except IndexError:
6   print 'Celsius degrees must be supplied on the command line'
7   sys.exit(1) # abort execution
8  except ValueError:
9   print 'Celsius degrees must be a pure number, '\
10        'not "%s"' % sys.argv[1]
11  sys.exit(1)
12
13 F = 9.0*C/5 + 32
14 print '%gC is %.1fF' % (C, F)
```

Exception handling (cont.)

Now, if we fail to provide a command-line argument, an `IndexError` occurs and we tell the user to write the `C` value on the command-line

```
1 Terminal > c2f_cml_except1.py 21C
2  Celsius degrees must be supplied on the command line
```

If the `float object` generation fails, because the CL has wrong syntax, a `ValueError` exception is raised and we branch into a second `EXCEPT block` and explain that the form of the given number is wrong

```
1 Terminal > c2f_cml_except1.py 21C
2  Celsius degrees must be a pure number, not "21C"
```

Exception handling (cont.)

Example

Some programming languages (Fortran, C, C++, and Perl, for example) allow list indices outside the legal index values

- Such unnoticed errors can be hard to find

```
1 >>> data = [1.0/i for i in range(1,10)]
2
3 >>> data[9]
4  ...
5  IndexError: list index out of range
```

Python always stops a program when an invalid index is encountered

- Unless the exception is handled explicitly, by the coder

Exception handling (cont.)

Example

Converting a `string object` into a `float object` fails and gives a `ValueError` exception, if the string is not an integer or a real number

```
>>> C = float('21 C')
 ...
 ValueError: invalid literal for float(): 21 C
```

Example

Trying to use an uninitialised variable gives a `NameError` exception

```
>>> print a
 ...
 NameError: name 'a' is not defined
```

Exception handling (cont.)

Example

Division by zero raises a `ZeroDivisionError` exception

```
>>> 3.0/0
 ...
 ZeroDivisionError: float division
```

Example

Writing a Python keyword illegally or performing a Python grammar error leads to a `SyntaxError` exception

```
>>> forr d in data:
 ...
 forr d in data:
      ^
 SyntaxError: invalid syntax
```

Exception handling (cont.)

Example

Try and multiply a string by a number

```
1 >>> 'a string'*3.14
2  ...
3  TypeError: can't multiply sequence by non-int of type 'float'
```

The `TypeError exception` is raised because the object types involved in the multiplication are incompatible (`str` and `float`)

Exception handling (cont.)

Example

```
>>> '--'*10 # ten double dashes = 20 dashes
    '--------------------'

>>> n = 4

>>> [1, 2, 3]*n
    [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> [0]*n
    [0, 0, 0, 0]
```

Raising exceptions
Handling errors

Raising exceptions

When an error occurs in your program, you can either print a message and use `sys.exit(1)` to abort execution, or you can raise an exception

- The latter (raising an exception) task is easy

Definition

Write `raise E(message)`, where `E` can be a known exception type and `message` refers to a string explaining what is wrong

Often `E` is

- `ValueError` if some variable value is wrong
- `TypeError` if some variable type is wrong

It is also possible to define own exception types

Exceptions can be raised from any place in a code

Raising exceptions (cont.)

Example

The next code shows how to test for exceptions and abort execution

```
1  try:
2   C = float(sys.argv[1])
3  except IndexError:
4   print 'Celsius degrees must be supplied on the command line'
5   sys.exit(1)  # abort execution
6  except ValueError:
7   print 'Celsius degrees must be a pure number, '\
8         'not "%s"' % sys.argv[1]
9   sys.exit(1)
10
11 F = 9.0*C/5 + 32
12 print '%gC is %.1fF' % (C, F)
```

Raising exceptions (cont.)

At times we see that an exception may not happen, if it happens ...

- We want a more precise error message to help the user

This can be done by raising a new exception in an `EXCEPT-block`

- To provide the desired exception type and message

Raising exceptions (cont.)

Another application of raising exceptions with tailored error messages arises when input data are invalid

Example

The reading of C and handling of errors are in a separate function

```
1  ##################################################################
2  def read_C():                                                    #
3   try:                                                            #
4    C = float(sys.argv[1])                                         #
5   except IndexError:                                              #
6    raise IndexError\                                              #
7    ('Celsius degrees must be supplied on command line')           #
8   except ValueError:                                              #
9    raise ValueError\                                              #
10   ('Celsius degrees must be a pure number, '\                    #
11    'not "%s"' % sys.argv[1])       # C is read correctly as a    #
12                                    # number, but can have       #
13                                    # wrong value                #
14  if C < -273.15:                                                 #
15   raise ValueError('C=%g is a non-physical value!' % C)          #
16  return C                                                        #
17 ##################################################################
```

There are two ways of using the read_C function

Raising exceptions (cont.)

The simplest of which is to call the function

```
1 C = read_C()
```

Wrong input will now lead to a raw dump of exceptions

```
1 Terminal > c2f_cml_v5.py
2  Traceback (most recent call last):
3   File "c2f_cml4.py", line 5, in ?
4    raise IndexError\
5 IndexError: Celsius degrees must be supplied on command line
```

New users get confused with raw output from exceptions, words like `Traceback`, `raise`, and `IndexError` make little sense to the newbie

Raising exceptions (cont.)

A more user-friendly output can be obtained by calling the `read_C function` inside `TRY-EXCEPT blocks`, check for `IndexError` or `ValueError`, and write the exception message in formatted form

This way, the programmer takes complete control of how the code behaves when errors are encountered

```
1 try:
2  C = read_C()
3 except Exception as e:
4  print e      # exception message
5  sys.exit(1) # terminate execution
```

Raising exceptions (cont.)

`Exception` is a name for all exceptions, `e` is an `exception object`

- Pretty print of the exception message from `print e`

To test more specifically for two exception types we can expect from `read_C`, instead of `Exception` we write (`ValueError`, `IndexError`)

```
1 try:
2  C = read_C()
3 except (ValueError, IndexError) as e:
4  print e      # exception message
5  sys.exit(1) # terminate execution
```

After the `TRY-EXCEPT blocks`, we do `F = 9*C/5 + 32` and print out `F`

Raising exceptions (cont.)

We test the program's behaviour when the input is wrong and right

```
1  Terminal > c2f_cml.py
2   Celsius degrees must be supplied on the command line
3
4  Terminal > c2f_cml.py 21C
5   Celsius degrees must be a pure number, not "21C"
6
7  Terminal > c2f_cml.py -500
8   C=-500 is a non-physical value!
9
10 Terminal > c2f_cml.py 21
11  21C is 69.8F
```

The program deals with wrong input by writing an informative message

- Execution is terminated, without annoying behaviour

Raising exceptions (cont.)

Scattered `IF-tests` with `sys.exit` calls are bad coding style

- Compared to the use of nested exception handling

Remark

It is a good practice to limit abortion in the main program only

The reason is that the functions can be re-used nevertheless

- The error can be dealt with differently

Remark

This coding style is considered the best way of dealing with errors, so we suggest to apply exceptions for handling potential errors in the code