Module 03 – Exception handling

Agenda

- Exceptions Intro
- Exceptions handling
- Argument of an Exception
- Raising Exceptions
- User-Defined Exceptions
- Logging
- Logging to a file
- Logging from multiple modules
- Logging variable data

Exceptions Intro

- An exception is an error that happens during the execution of a program. When that error occurs, Python generates an exception that can be handled
- Unhandled exceptions cause your program to crash.
- Exceptions come in different types. Python generates an exception with type, suitable to an error
- The words "try", except" and "finally" are Python keywords that used to catch exceptions.
- Exceptions can be raised (thrown) using "raise" statement

Exceptions Intro — cont’d

For Example:

```
try:
        print(1/0)
except ZeroDivisionError:
        print ("You can't divide by zero")
```

Some Built-in Exception Errors

- Below is some common exceptions errors in Python:
  - **Exception** Base class for all exceptions
  - **lOError** If the file cannot be opened.
  - **ImportError** If python cannot find the module
  - **ValueError** Raised when built-in function receives an inappropriate value
  - **KeyError** Raised when the specified key is not found in the dictionary.
  - **NameError** Raised when an identifier is not found in the local or global namespace.
  - **EOFError** Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data
  - **ArithmeticError** The base exceptions for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError
  - **SyntaxError** Raised when the syntax is incorrect

Exception handling example

Demo

Version 1:

```
s = input("Enter a number between 1 - 10")
   number = int(s)
   print ("your number is — {}".format(number))
```

- If the input is not numeric, "67a" for example, the code — raises exception:

  ValueError: invalid literal for int() with base 10: '67a'

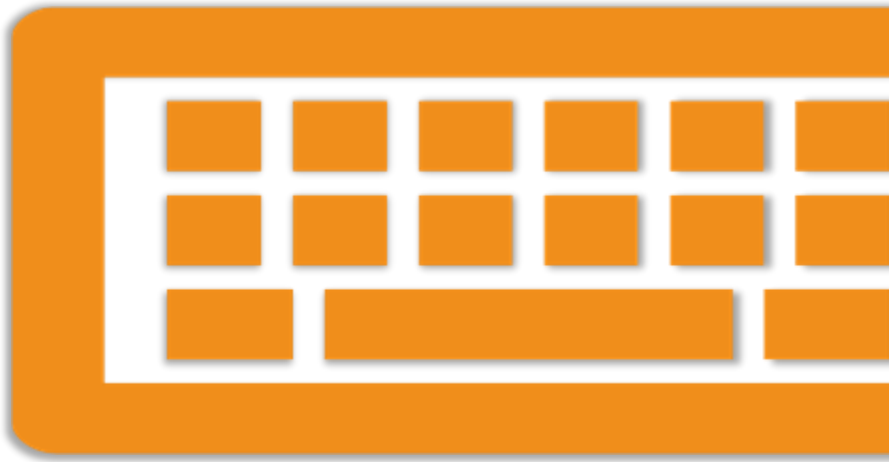
Version 2:

```
number =5
try:
   s = input("Enter a number between 1-10, 5 is         default")
   number = int(s)
except ValueError:
   print ("your value is incorrect, default value is set")
print ("your number is — {}".format(number))
```

- The program will continue running whether the input is correct or not.

Lab 01

Lab



Argument of an Exception

- Anexception can have an argument, that holds an additional information about the problem and cause of the exception

```
try:
        print(val + 10)
except NameError as ne:
        print(ne)
```

File "<stdin>", line 1, in <module>

NameError: name '‘val' is not defined

Multiple excepts

try statement supports multiple excepts

```
try:
        ……
except ValueError as ve:
        ……
except ArithmeticError as ae:
        ……
except Exception:
        ……
```

Try ... finally example

```
fo = open("some file","r")
try:
        # work with file
finally:
    fo.close()
```

Raising Exceptions

- The raise statement allows the programmer to force a specinie exception to occur

- raise can come with a parameter (or several parameters). The parameter is a message or any additional information about the error

- raise usually placed in infrastructure methods, classes and modules.

Raising Exceptions

Demo

Raising Exceptions

```
def check_grade_validation(grade_value):
    if type(grade_value) != int:
        raise TypeError("grade value must be integer")
    if grade_value < 0 or grade_value>100:
        raise ValueError("grade value must be between O — 100")
```

User-Defined Exceptions

- Python has many built-in exceptions
- Sometimes we may need to create a custom exceptions that serves our purpose.
- Python, supports creation of a custom exception by defining a new class that derives from Exception built in class
- Usually custom exceptions add some additional information

User-Defined Exceptions

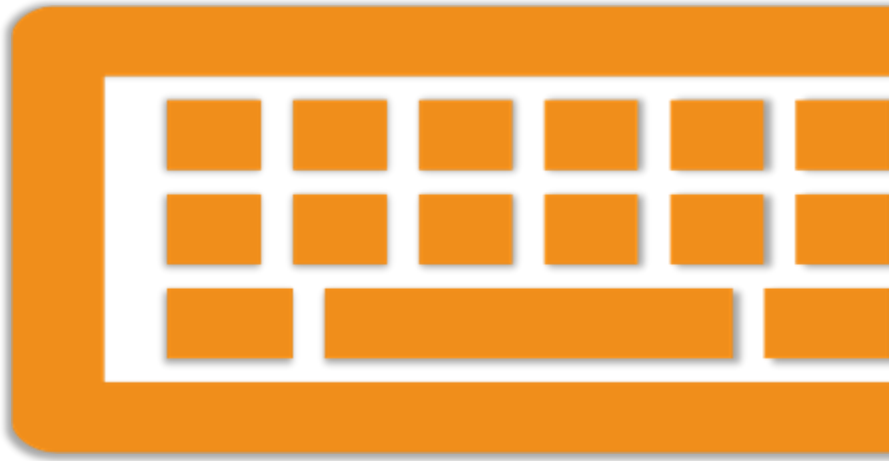
Demo

User-Defined Exceptions

```
class ValidationError(Exception):
    def __init__(self, message, errors):
        Exception.__init__(self, message) # base class __init__
        self.errors = errors
    def __str__(self):
        return "{}, error is {}".format(Exception.__str__(self) ,self.error)
```

Lab 02

Lab



Logging

- Logging is a means of tracking events that happen when some software runs.
- The software's developer adds logging calls to their code to indicate that certain events have occurred.
- An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event).
- Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

When to use logging?

- Logging provides a set of convenience functions for simple logging usage. These are debug(), info(), warning(), error() and critical(). To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	print()
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	logging.info() (or logging.debug() for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	warnings.warn() in library code if the issue is avoidable and the client application should be modified to eliminate the warning logging.warning() if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	logging.error(), logging.exception() or logging.critical() as appropriate for the specific error and application domain

When to use logging?

- The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

Logging simple example

Demo

Example

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

Logging to a file

- A very common situation is that of recording logging events in a file, so let's look at that next. Be sure to try the following in a newly started Python interpreter, and don't just continue from the session described above:

```
import logging
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

- The encoding argument was added. In earlier Python versions, or if not specified, the encoding used is the default value used by open(). While not shown in the above example, an errors argument can also now be passed, which determines how encoding errors are handled. For available values and the default, see the documentation for open().

And now if we open the file and look at what we have, we should find the log messages:

DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
ERROR:root:And non-ASCII stuff, too, like Øresund and Malmö

Logging to a file – cont'd

- To get the value which you'll pass to **basicConfig()** via the level argument. You may want to error check any user input value, perhaps as in the following example:

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

- The call to basicConfig() should come before any calls to debug(), info(), etc. Otherwise, those functions will call basicConfig() for you with the default options. As it's intended as a one-off simple configuration facility, only the first call will actually do anything: subsequent calls are effectively no-ops.

Logging to a file – cont’d

- If you run the above script several times, the messages from successive runs are appended to the file *example.log*. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the *filemode* argument, by changing the call in the above example to:

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

- The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.

Logging from multiple modules

- If your program consists of multiple modules, here's an example of how you could organize logging in it:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

Logging from multiple modules – cont'd

- If you run myapp.py, you should see this in myapp.log:

  ```
  INFO:root:Started
  INFO:root:Doing something
  INFO:root:Finished
  ```

- Which is hopefully what you were expecting to see. You can generalize this to multiple modules, using the pattern in mylib.py. Note that for this simple usage pattern, you won't know, by looking in the log file, where in your application your messages came from, apart from looking at the event description.

Logging variable data

- To log variable data, use a format string for the event description message and append the variable data as arguments. For example:

```
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

will display:

WARNING:root:Look before you leap!

- As you can see, merging of variable data into the event description message uses the old, %-style of string formatting. This is for backwards compatibility: the logging package pre-dates newer formatting options such as str.format() and string.Template.

Changing the format of displayed messages

- To change the format which is used to display messages, you need to specify the format you want to use:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s',
level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

which would print:

DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too

Displaying the date/time in messages

- To display the date and time of an event, you would place '%(asctime)s' in your format string:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

which should print something like this:

2010-12-12 11:41:42,612 is when this event was logged.

- The default format for date/time display (shown above) is like ISO8601 or RFC 3339. If you need more control over the formatting of the date/time, provide a **datefmt** argument to **basicConfig**, as in this example:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

which would display something like this:

12/12/2010 11:46:36 AM is when this event was logged.

Questions