Python

Matt Waismann

Sep 20th, 2021

1 Logging

Without logging, most people check the program behaved correctly by embedding print statements at various points in the script. The better practice is to use *logging*.

There are five standard logging levels. The levels are ordered by severity:

- # 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 functionality can be accessed through the standard library 'logging'

```
import logging
```

The default level for logging is set to 'WARNING'. This means it will capture everything that is everything at level 'WARNING' and above. Therefore, it includes 'WARNING', 'ERROR', and 'CRITICAL'. The default logging behaviour is to just log results to the console. If we want to log levels below WARNING we need to modify the default behavior. Here's a logging example:

```
add_result = add(num_1, num_2)
logging.warning('Add: {} + {} = {}'.format(num_1, num_2, add_result))
```

Output in the console will look like:

```
WARNING:root:Add: 10 + 5 = 15
```

Of course, this isn't a warning so lets instead change the default logging level

```
logging.basicConfig(level = logging.DEBUG)
```

Let's now change the logging output level to something more appropriate like 'DEBUG'

```
add_result = add(num_1, num_2)
logging.warning('Add: {} + {} = {}'.format(num_1, num_2, add_result))
```

As said earlier, default logging only outputs to the console. If you prefer to have a log file, include a file name in the logging.basicConfig mutator

```
logging.basicConfig(filename = 'test.log', level = logging.DEBUG)
```

The default log format is to include logging level is

```
LOGLEVEL:logger:message (e.g. DEBUG:root:Add: 10 + 5 = 15 )
```

To modify the log format to include things like date and time, include a 'format' argument in the logging.basicConfig mutator.

```
logging.basicConfig(filename = 'test.log', level = logging.DEBUG,
    format = '%(asctime)s:%(levelname)s:%(message)s')
```

Now the output in the log file will look like

```
2021-09-20 12:53:08,118: DEBUG: Add: 10 + 5 = 20
```

A sample program might look like this

```
import logging
logging.basicConfig(filename='employee.log', level = logging.INFO,
                    format= '%(levelname)s:%(message)s')
class Employee:
  def __init__(self, first, last):
      self.first = first
      self.last = last

      logging.info('Created Employee: {} - {}'. format(self.fullname,
          self.email))
  @property
  def email(self):
      return '{}.{}@email.com'.format(self.first, self.last)
  @property
  def fullname(self):
```

```
        return '{} {}'.format(self.first, self.last)

emp_1 = Employee('John', 'Smith')
emp_2 = Employee('Jane', 'Doe')
```

The 'employee.log' file should look like

```
INFO:Created Employee: John Smith - John.Smith@gmail.com
INFO:Created Employee: Jane Doe - Jane.Doe@gmail.com
```

2 Error Handling

A common type of error in Python is when a certain expression is not formed in inaccordance with the prescribed usage. Such an error is called a syntax error. Syntax errors are detected before the program is executed.

```
>>> print "hello world"
SyntaxError: Missing parentheses in call to 'print'. Did you mean
    print("hello")?
```

Here we will be concerned with a different type of error, a runtime error, called an exception. This is type of error is detected by the Python interpreter while the program is executing. When these errors occur, Python writes a Traceback to the console, specificying the path to where the interpreter detected the error and the name of the error (e.g. FileNoteFoundError).

The fundemental building blocks of error handling are

```
try:
    pass
except Exception:
    pass
else:
    pass
finally:
    pass
```

The Python interpreter will execute whatever code is in the try statement. If an exception occurs, instead of the program stopping, we enter the except clause. The generalized exception is 'Exception'. This will capture any exception that occurs in the try block. However, this is not a best practice since the motivation for error handling is to address specific errors that occur. We will only enter the except clause if no exception is specified, the generalized 'Exception' is used, or the correct exception expression is used.

Note that we can include multiple except clauses. The Python interpreter

will attempt each except statement until it finds one that allows the exception to enter (the three cases described earlier) then it will not attempt to enter any other except statements. Therefore, it's best practice to include the most specific except statements at the top and the most general exceptions (if you want them) at the bottom.

The *as* clause can be used to assign an alias to an exception name in the except statement. This is one of the three statements in Python which permit the use of *as*. The other two cases are in *import* statements and the *with* statement to name the resource being allocated. Here we see the usage for except statements:

```
try:
  f = open('test_file.txt')
except FileNotFoundError as e:
  print(e)
```

The *else* clause executes code if the *try* clause doesn't raise an exception.

The *finally* clause executes regardless of what happens, exception or not. This has a common use case of closing a connection to a database.

Note that we have the ability to raise exceptions ourself with the use of the *raise* clause. This typically is used inside of an if statement where a certain failure criteria was met. We can use the *raise* clause within a try statement or even outside the context of exception hanlding statements.

Here's an example that summarizes everything we've discussed:

```
try:
  f = open('my_data.txt')
  if f.name == "not_my_data.txt':
      raise Exception("This is the wrong file")
      raise ValueError("This is just to show that you can raise
          different types of exceptions")
  except FileNoteFoundError as e:
      print(e)
  except Exception:
      print("something else went wrong")
  else:
      print(f.read())
      f.close()
  finally:
      print("The code executed but it may or may not have contained an
          exception. I am the finally statement so I execute
          regardless.")
```

2.1 Creating Your Own Exception

To create your own exception you must create a custom exception class that inherits from the Exception class. The structure should be as follow:

```
class CoffeeTooHotException(Exception):
    def __init__(self, msg):
        super().__init__(msg)
```

Now you can raise a new exception like any other exception:

```
if self.__temperature > 85:
    raise CoffeeTooHotException("Coffee too hot")
```