3.1 Advanced techniques of creating and serving exceptions

In this module, we'll talk about Python exceptions – objects that represent errors which occur during the execution of a program that disrupts the normal flow of the program's instructions.

Plan for the module:

- short introduction to exceptions;
- review of the named attributes of exception objects;
- introduction to chained exceptions;
- analysis of the traceback object of each exception.

Exceptions - short introduction

When Python executes a script and encounters a situation that it cannot cope with, it:

- stops your program;
- creates a special kind of data, called an **exception**. Of course, this exception is an object.

Both of these activities are called **raising an exception**. We can say that Python always raises an exception (or that an exception has been raised) when it has no idea what do to with your code.

What happens next?

- the raised exception expects somebody or something to notice it and take care of it;
- if nothing happens to take care of the raised exception, the program will be **forcibly terminated**, and you will see an error message sent to the console by Python;
- otherwise, if the exception is taken care of and **handled** properly, the suspended program can be resumed and its execution can continue.

Python provides effective tools that allow you to **observe exceptions, identify them and handle them efficiently**. This is possible due to the fact that all potential exceptions have their unambiguous names, so you can categorize them and react appropriately.

You may already have met an exception error message like the following:

output

```
IndexError: list index out of range
```

Sooner or later, every Pythonista will write a code that raises an exception, and that is why it is so important to know how to deal with exceptions.

Python comes with 63[1)] built-in exceptions, and they can be represented in the form of a tree-shaped hierarchy. The reason for this is that exceptions are inherited from BaseException, the most general exception class.

And this approach tells you that you can also create your own specific exception classes – the only constraint is: you have to subclass BaseException or any other derived exception class.

Exception handling

When you suspect that the code may raise an exception, you should use the `try: problematic_code` except code block to surround the «problematic» piece of code. In effect, when the exception is raised, execution is not terminated, but the code following the `except` clause will try to handle the problem in an elegant way.

Let's look at typical try ... except statement.

```
try:
    print(int('a'))
except ValueError:
    print('You tried to do a nasty thing...')
```

Whenever you try to convert letter 'a' to an integer value, you'll spot an exception. In the case where you get data from an external source (console, file, etc.) you should not trust the data types, so it’s wise to surround the fragile code (int() in this example) with a try... except block.

Advanced exceptions - named attributes

Now it’s time to dive more deeply into the interesting features of exception concepts and get familiar with the possible ways of using those concepts in your applications.

Once again, let's look at a typical try ... except statement.

```
try:
    print(int('a'))
except ValueError as e_variable:
    print(e_variable.args)
```

The except clause may specify a variable after the exception name. In this example it’s an `e_variable`. This variable is bound to an exception instance with the arguments stored in the `args` attribute of the **e_variable** object.

Some exception objects carry additional information about the exception itself.

The `ImportError` exception – raised when the import statement has trouble trying to load a module. The attributes are:

- name – represents the name of the module that was attempted to be imported;
- path – represents the path to any file which triggered the exception, respectively. Could be None.

See the output of this snippet to analyze the attributes' values.

```
try:
    import abcdefghijk

except ImportError as e:
    print(e.args)
    print(e.name)
    print(e.path)
```

output

```
("No module named 'abcdefghijk'",)
abcdefghijk
None
```

The `UnicodeError` exception – raised when a Unicode-related encoding or decoding error occurs. It is a subclass of ValueError.

The UnicodeError has attributes that describe an encoding or decoding error.

- **encoding** – the name of the encoding that raised the error.
- **reason** – a string describing the specific codec error.
- **object** – the object the codec was attempting to encode or decode.
- **start** – the first index of invalid data in the object.
- **end** – the index after the last invalid data in the object.

See the output of this snippet to analyze the attributes' values.

```
try:
    b'\x80'.decode("utf-8")
except UnicodeError as e:
    print(e)
    print(e.encoding)
    print(e.reason)
    print(e.object)
    print(e.start)
    print(e.end)
```

output

```
'utf-8' codec can't decode byte 0x80 in position 0: invalid start byte
utf-8
invalid start byte
b'\x80'
0
1
```

What are chained exceptions?

Python 3 introduced a very interesting feature called 'Exception chaining' to effectively deal with exceptions.

Imagine a situation where you are already handling an exception and your code incidentally triggers an additional exception. Should your code lose the information about the previous exception? Of course not. So the information should be available to the code following the erroneous code. This is an example of implicit chaining.

Another case pops up when we knowingly wish to handle an exception and translate it to another type of exception. Such a situation is typical when you have a good reason for the unifying behavior of one piece of code to act similarly to another piece of code, like a legacy code. In this situation it would also be nice to keep the details of the former exception. This is an example of explicit chaining.

This chaining concept introduces two attributes of exception instances:

the `__context__` attribute, which is inherent for implicitly chained exceptions; the `__cause__` attribute, which is inherent for explicitly chained exceptions. Those attributes help the programmer to keep a reference to the original exception object in a handy and consistent way for later processing like logging, etc.

Look at the following code and the output traceback. Pay attention to the fact that we are not raising any exception explicitly with a raise statement, but we cause it implicitly (BTW: dividing by 0 is always a good way to cause an exception or error in most of the programming languages):

output

```
Traceback (most recent call last):
  File "exceptions#030.py", line 6, in <module>
    print(a_list[3])
IndexError: list index out of range

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "exceptions#030.py", line 8, in <module>
    print(1 / 0)
ZeroDivisionError: division by zero
```

```
a_list = ['First error', 'Second error']

try:
    print(a_list[3])
except Exception as e:
    print(0 / 0)
```

The original exception objecte is now being referenced by the `__context__` attribute of the following exception `f`.

output

```
Inner exception (f): division by zero
Outer exception (e): list index out of range
Outer exception referenced: list index out of range
Is it the same object: True
```

The `except Exception` clause is a wide one and normally should be used as a last resort to catch all unhandled exceptions. It’s so wide because we don’t know what kind of exception might occur.

So, when a subsequent exception (much better forecasted) occurs, we still can say a lot about the nature of the first exception.

Isn't it handled with ease?

```
a_list = ['First error', 'Second error']

try:
    print(a_list[3])
except Exception as e:
    try:
        # the following line is a developer mistake - they wanted to print progress
```

```
as 1/10   but wrote 1/0
        print(1 / 0)
    except ZeroDivisionError as f:
        print('Inner exception (f):', f)
        print('Outer exception (e):', e)
        print('Outer exception referenced:', f.__context__)
        print('Is it the same object:', f.__context__ is e)
```

output

```
Inner exception (f): division by zero
Outer exception (e): list index out of range
Outer exception referenced: list index out of range
Is it the same object: True
```

Advanced exceptions - explicitly chained exceptions

This time we'd like to convert an explicit type of exception object to another type of exception object **at the moment when the second exception is occurring**.

Imagine that your code is responsible for the final checking process before the rocket is launched. The list of checks is a long one, and different checks could result in different exceptions.

But as it is a very serious process, you should be sure that all checks are passed. If any fails, it should be marked in the log book and re-checked next time.

Now you see that it would be convenient to convert each type of exception into its own exception (like RocketNotReadyError) and to log the origin of the exception.

output

```
Final check procedure
        The captain's name is John
        The pilot's name is Mary
        The mechanic's name is Mike
Traceback (most recent call last):
  File "exceptions#050.py", line 10, in personnel_check
    print("\tThe navigator's name is", crew[3])
IndexError: list index out of range

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "exceptions#050.py", line 19, in
    personnel_check()
  File "exceptions#050.py", line 12, in personnel_check
    raise RocketNotReadyError('Crew is incomplete') from e
__main__.RocketNotReadyError: Crew is incomplete
```

```
class RocketNotReadyError(Exception):
    pass
```

```
def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e

crew = ['John', 'Mary', 'Mike']
print('Final check procedure')

personnel_check()
```

To catch the cause of the **RocketNotReadyError** exception, you should access the `__cause__` attribute of the **RocketNotReadyError** object.

Run the code in the right pane and examine the output.

This time the report is handled in a safe way, and you can be sure that you're doing a good job. Once again, the result of the code execution contains an interesting piece of information indicating that we have just witnessed a chain of exceptions:

output

```
Final check procedure
        The captain's name is John
        The pilot's name is Mary
        The mechanic's name is Mike
General exception: "Crew is incomplete", caused by "list index out of range"
```

```
class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e

crew = ['John', 'Mary', 'Mike']
print('Final check procedure')

try:
    personnel_check()
except RocketNotReadyError as f:
    print('General exception: "{}", caused by "{}"'.format(f, f.__cause__))
```

Have a look at an extended checklist script.

```
class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e


def fuel_check():
    try:
        print('Fuel tank is full in {}%'.format(100 / 0))
    except ZeroDivisionError as e:
        raise RocketNotReadyError('Problem with fuel gauge') from e


crew = ['John', 'Mary', 'Mike']
fuel = 100
check_list = [personnel_check, fuel_check]

print('Final check procedure')

for check in check_list:
    try:
        check()
    except RocketNotReadyError as f:
        print('RocketNotReady exception: "{}", caused by "{}"'.format(f,
f.__cause__))
```

Pay attention to the fact that thanks to polymorphism and explicit chaining, our approach has become more generic: we are able to run two different checks, each returning a different exception type.

And we're still able to handle them correctly, as we're hiding some details behind the RocketNotReadyError exception object.

output

```
        The captain's name is John
        The pilot's name is Mary
        The mechanic's name is Mike
RocketNotReady exception: "Crew is incomplete", caused by "list index out of
range"
RocketNotReady exception: "Problem with fuel gauge", caused by "division by
zero"
```

LAB

Scenario

- Try to extend the check list script to handle more different checks, all reported as RocketNotReady exceptions.
- Add your own checks for: batteries and circuits.

- ```
  class RocketNotReadyError(Exception):
      pass


  def personnel_check():
      try:
          print("\tThe captain's name is", crew[0])
          print("\tThe pilot's name is", crew[1])
          print("\tThe mechanic's name is", crew[2])
          print("\tThe navigator's name is", crew[3])
      except IndexError as e:
          raise RocketNotReadyError('Crew is incomplete') from e


  def fuel_check():
      try:
          print('Fuel tank is full in {}%'.format(100/0))
      except ZeroDivisionError as e:
          raise RocketNotReadyError('Problem with fuel gauge') from e

  def batteries_check():
      # add your own implementation
      pass

  def circuits_check():
      # add your own implementation
      pass


  crew = ['John', 'Mary', 'Mike']
  fuel = 100
  check_list = [personnel_check, fuel_check, batteries_check, circuits_check]

  print('Final check procedure')

  for check in check_list:
      try:
          check()
      except RocketNotReadyError as f:
          print('RocketNotReady exception: "{}", caused by "{}"'.format(f,
  f.__cause__))
  ```

Advanced exceptions - the traceback attribute

Each exception object owns a `__traceback__` attribute.

Python allows us to operate on the traceback details because each exception object (not only chained ones)
  ```

owns a `__traceback__` attribute.

Let's examine such an object while we're preparing our rocket to launch.ç

```
class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e


crew = ['John', 'Mary', 'Mike']

print('Final check procedure')

try:
    personnel_check()
except RocketNotReadyError as f:
    print(f.__traceback__)
    print(type(f.__traceback__))
```

```
  <code ; output>
```

Final check procedure

```
      The captain's name is John
      The pilot's name is Mary
      The mechanic's name is Mike
```

<traceback object at 0x00753300> <class 'traceback'> </code>

From the output presented on the previous page, we can conclude that we have to deal with a traceback type object.

To achieve this, we could use the `format_tb()` method delivered by the built-in traceback module to get a list of strings describing the traceback.

We could use the `print_tb()` method, also delivered by the traceback module, to print strings directly to the standard output.

The corresponding output reveals the sequence of exceptions and proves that the execution was not interrupted by the exceptions because we see the final wording **Final check is over**.

```
import traceback

class RocketNotReadyError(Exception):
    pass
```

```
def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e


crew = ['John', 'Mary', 'Mike']

print('Final check procedure')

try:
    personnel_check()
except RocketNotReadyError as f:
    print(f.__traceback__)
    print(type(f.__traceback__))
    print('\nTraceback details')
    details = traceback.format_tb(f.__traceback__)
    print("\n".join(details))

print('Final check is over')
```

output

```
Final check procedure
        The captain's name is John
        The pilot's name is Mary
        The mechanic's name is Mike
<traceback object at 0x00D373A0>
<class 'traceback'>

Traceback details
  File "exceptions#090.py", line 22, in <module>
    personnel_check()

  File "exceptions#090.py", line 14, in personnel_check
    raise RocketNotReadyError('Crew is incomplete') from e

Final check is over
```

Now your log book could be filled with lots of details about your rocket launch for later investigation.

In real life development projects, you may make use of logged tracebacks after comprehensive test sessions to gather statistics or even automate bug reporting processes.

For more information about chained exceptions and traceback attributes, look at the PEP 3134 document.

1)
This number may vary across different Python versions.