Exceptions once again

More about exceptions

Discussing object programming offers a very good opportunity to return to exceptions. The objective nature of Python's exceptions makes them a very flexible tool, able to fit to specific needs, even those you don't yet know about.

Before we dive into the **objective face of exceptions**, we want to show you some syntactical and semantic aspects of the way in which Python treats the try-except block, as it offers a little more than what we have presented so far.

The first feature we want discuss here is an additional, possible branch that can be placed inside (or rather, directly behind) the try-except block - it's the part of the code starting with `else` - just like in the example below.

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        return None
    else:
        print("Everything went fine")
        return n

print(reciprocal(2))
print(reciprocal(0))
```

A code labelled in this way is executed when (and only when) no exception has been raised inside the `try:` part. We can say that exactly one branch can be executed after `try:` - either the one beginning with `except` (don't forget that there can be more than one branch of this kind) or the one starting with `else`.

Note: the `else:` branch has to be located after the last `except` branch.

The example code produces the following output:

```
Everything went fine
0.5
Division failed
None
```

The try-except block can be extended in one more way - by adding a part headed by the `finally` keyword (it must be the last branch of the code designed to handle exceptions).

Note: these two variants (`else` and `finally`) aren't dependent in any way, and they can coexist or occur independently.

The `finally` block is always executed (it finalizes the try-except block execution, hence its name), no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not.

Look at the code.

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        n = None
    else:
        print("Everything went fine")
    finally:
        print("It's time to say goodbye")
        return n

print(reciprocal(2))
print(reciprocal(0))
```

It outputs:

```
Everything went fine
It's time to say good bye
0.5
Division failed
It's time to say good bye
None
```

Exceptions are classes

All the previous examples were content with detecting a specific kind of exception and responding to it in an appropriate way. Now we're going to delve deeper, and look inside the exception itself.

You probably won't be surprised to learn that **exceptions are classes**. Furthermore, when an exception is raised, an object of the class is instantiated, and goes through all levels of program execution, looking for the except branch that is prepared to deal with it.

Such an object carries some useful information which can help you to precisely identify all aspects of the pending situation. To achieve that goal, Python offers a special variant of the exception clause - you can find it below.

```
try:
    i = int("Hello!")
except Exception as e:
    print(e)
    print(e.__str__())
```

As you can see, the `except` statement is extended, and contains an additional phrase starting with the `as` keyword, followed by an identifier. The identifier is designed to catch the exception object so you can analyze its nature and draw proper conclusions.

Note: the identifier's scope covers its `except` branch, and doesn't go any further.

The example presents a very simple way of utilizing the received object - just print it out (as you can see, the output is produced by the object's `__str__()` method) and it contains a brief message describing the reason.

The same message will be printed if there is no fitting `except` block in the code, and Python is forced to handle it alone.

All the built-in Python exceptions form a hierarchy of classes. There is no obstacle to extending it if you find it reasonable.

Look at the code.

```
def printExcTree(thisclass, nest = 0):
    if nest > 1:
        print("   |" * (nest - 1), end="")
    if nest > 0:
        print("   +---", end="")

    print(thisclass.__name__)

    for subclass in thisclass.__subclasses__():
        printExcTree(subclass, nest + 1)

printExcTree(BaseException)
```

This program dumps all predefined exception classes in the form of a tree-like printout.

As **a tree is a perfect example of a recursive data structure**, a recursion seems to be the best tool to traverse through it. The `printExcTree()` function takes two arguments:

- a point inside the tree from which we start traversing the tree;
- a nesting level (we'll use it to build a simplified drawing of the tree's branches)

Let's start from the tree's root - the root of Python's exception classes is the `BaseException` class (it's a superclass of all other exceptions).

For each of the encountered classes, perform the same set of operations:

- print its name, taken from the `__name__` property;
- iterate through the list of subclasses delivered by the `__subclasses__()` method, and recursively invoke the `printExcTree()` function, incrementing the nesting level respectively.

  Note how we've drawn the branches and forks. The printout isn't sorted in any way - you can try to sort it yourself, if you want a challenge. Moreover, there are some subtle inaccuracies in the way in which some branches are presented. That can be fixed, too, if you wish.

This is how it looks:

```
BaseException
      +---Exception
      | +---TypeError
      | +---StopAsyncIteration
      | +---StopIteration
      | +---ImportError
      | | +---ModuleNotFoundError
      | | +---ZipImportError
      | +---OSError
      | | +---ConnectionError
      | | | +---BrokenPipeError
      | | | +---ConnectionAbortedError
      | | | +---ConnectionRefusedError
      | | | +---ConnectionResetError
      | | +---BlockingIOError
      | | +---ChildProcessError
      | | +---FileExistsError
      | | +---FileNotFoundError
      | | +---IsADirectoryError
      | | +---NotADirectoryError
      | | +---InterruptedError
      | | +---PermissionError
      | | +---ProcessLookupError
      | | +---TimeoutError
      | | +---UnsupportedOperation
      | | +---herror
      | | +---gaierror
      | | +---timeout
      | | +---Error
      | | | +---SameFileError
      | | +---SpecialFileError
      | | +---ExecError
      | | +---ReadError
      | +---EOFError
      | +---RuntimeError
      | | +---RecursionError
      | | +---NotImplementedError
      | | +---_DeadlockError
      | | +---BrokenBarrierError
      | +---NameError
      | | +---UnboundLocalError
      | +---AttributeError
      | +---SyntaxError
      | | +---IndentationError
      | | | +---TabError
      | +---LookupError
      | | +---IndexError
      | | +---KeyError
      | | +---CodecRegistryError
      | +---ValueError
      | | +---UnicodeError
      | | | +---UnicodeEncodeError
```

```
| | | +---UnicodeDecodeError
| | | +---UnicodeTranslateError
| | +---UnsupportedOperation
| +---AssertionError
| +---ArithmeticError
| | +---FloatingPointError
| | +---OverflowError
| | +---ZeroDivisionError
| +---SystemError
| | +---CodecRegistryError
| +---ReferenceError
| +---BufferError
| +---MemoryError
| +---Warning
| | +---UserWarning
| | +---DeprecationWarning
| | +---PendingDeprecationWarning
| | +---SyntaxWarning
| | +---RuntimeWarning
| | +---FutureWarning
| | +---ImportWarning
| | +---UnicodeWarning
| | +---BytesWarning
| | +---ResourceWarning
| +---error
| +---Verbose
| +---Error
| +---TokenError
| +---StopTokenizing
| +---Empty
| +---Full
| +---_OptionError
| +---TclError
| +---SubprocessError
| | +---CalledProcessError
| | +---TimeoutExpired
| +---Error
| | +---NoSectionError
| | +---DuplicateSectionError
| | +---DuplicateOptionError
| | +---NoOptionError
| | +---InterpolationError
| | | +---InterpolationMissingOptionError
| | | +---InterpolationSyntaxError
| | | +---InterpolationDepthError
| | +---ParsingError
| | | +---MissingSectionHeaderError
| +---InvalidConfigType
| +---InvalidConfigSet
| +---InvalidFgBg
| +---InvalidTheme
| +---EndOfBlock
| +---BdbQuit
| +---error
```

```
| +---_Stop
| +---PickleError
| | +---PicklingError
| | +---UnpicklingError
| +---_GiveupOnSendfile
| +---error
| +---LZMAError
| +---RegistryError
| +---ErrorDuringImport
+---GeneratorExit
+---SystemExit
+---KeyboardInterrupt
```

Detailed anatomy of exceptions

Let's take a closer look at the exception's object, as there are some really interesting elements here (we'll return to the issue soon when we consider Python's input/output base techniques, as their exception subsystem extends these objects a bit).

The `BaseException` class introduces a property named `args`. It's a **tuple designed to gather all arguments passed to the class constructor**. It is empty if the construct has been invoked without any arguments, or contains just one element when the constructor gets one argument (we don't count the `self` argument here), and so on.

We've prepared a simple function to print the `args` property in an elegant way. You can see the function below.

```python
def printargs(args):
    lng = len(args)
    if lng == 0:
        print("")
    elif lng == 1:
        print(args[0])
    else:
        print(str(args))

try:
    raise Exception
except Exception as e:
    print(e, e.__str__(), sep=' : ' ,end=' : ')
    printargs(e.args)

try:
    raise Exception("my exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    printargs(e.args)

try:
    raise Exception("my", "exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    printargs(e.args)
```

We've used the function to print the contents of the `args` property in three different cases, where the exception of the `Exception` class is raised in three different ways. To make it more spectacular, we've also printed the object itself, along with the result of the `__str__()` invocation.

The first case looks routine - there is just the name `Exception` after the `raise` keyword. This means that the object of this class has been created in a most routine way.

The second and third cases may look a bit weird at first glance, but there's nothing odd here - these are just the constructor invocations. In the second `raise` statement, the constructor is invoked with one argument, and in the third, with two.

As you can see, the program output reflects this, showing the appropriate contents of the `args` property:

```
     : :
my exception : my exception : my exception
('my', 'exception') : ('my', 'exception') : ('my', 'exception')
```

How to create your own exception

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions.

It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python.

This is done by **defining your own, new exceptions as subclasses derived from predefined ones**.

Note: if you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one. If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.

Imagine that you've created a brand new arithmetic, ruled by your own laws and theorems. It's clear that division has been redefined, too, and has to behave in a different way than routine dividing. It's also clear that this new division should raise its own exception, different from the built-in `ZeroDivisionError`, but it's reasonable to assume that in some circumstances, you (or your arithmetic's user) may want to treat all zero divisions in the same way.

Demands like these may be fulfilled in the way presented in the editor. Look at the code.

```
class MyZeroDivisionError(ZeroDivisionError):
    pass

def doTheDivision(mine):
    if mine:
        raise MyZeroDivisionError("some worse news")
    else:
        raise ZeroDivisionError("some bad news")

for mode in [False, True]:
    try:
        doTheDivision(mode)
    except ZeroDivisionError:
        print('Division by zero')


for mode in [False, True]:
    try:
        doTheDivision(mode)
    except MyZeroDivisionError:
        print('My division by zero')
    except ZeroDivisionError:
        print('Original division by zero')
```

And let's analyze it:

- We've defined our own exception, named `MyZeroDivisionError`, derived from the built-in `ZeroDivisionError`. As you can see, we've decided not to add any new components to the class.

  In effect, an exception of this class can be - depending on the desired point of view - treated like a plain `ZeroDivisionError`, or considered separately.

- The `doTheDivision()` function raises either a `MyZeroDivisionError` or `ZeroDivisionError` exception, depending on the argument's value.

  The function is invoked four times in total, while the first two invocations are handled using only one `except` branch (the more general one) and the last two ones with two different branches, able to distinguish the exceptions (don't forget: the order of the branches makes a fundamental difference!)

When you're going to build a completely new universe filled with completely new creatures that have nothing in common with all the familiar things, you may want to **build your own exception structure**.

For example, if you work on a large simulation system which is intended to model the activities of a pizza restaurant, it can be desirable to form a separate hierarchy of exceptions.

You can start building it by **defining a general exception as a new base class** for any other specialized exception. We've done in in the following way:

```
class PizzaError(Exception):
```

```
    def __init__(self, pizza, message):
        Exception.__init__(message)
        self.pizza = pizza
```

Note: we're going to collect more specific information here than a regular `Exception` does, so our constructor will take two arguments:

- one specifying a pizza as a subject of the process,
- and one containing a more or less precise description of the problem.

As you can see, we pass the second parameter to the superclass constructor, and save the first inside our own property.

A more specific problem (like an excess of cheese) can require a more specific exception. It's possible to derive the new class from the already defined `PizzaError` class, like we've done here:

```
class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError._init__(self, pizza, message)
        self.cheese = cheese
```

The `TooMuchCheeseError` exception needs more information than the regular `PizzaError` exception, so we add it to the constructor - the name `cheese` is then stored for further processing.

Look at the code.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def makePizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no such pizza on the menu")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "too much cheese")
    print("Pizza ready!")


for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        makePizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

We've coupled together the two previously defined exceptions and harnessed them to work in a small example snippet.

One of these is raised inside the `makePizza()` function when any of these two erroneous situations is discovered: a wrong pizza request, or a request for too much cheese.

Note:

- removing the branch starting with `except TooMuchCheeseError` will cause all appearing exceptions to be classified as `PizzaError`;
- removing the branch starting with `except PizzaErrorwill` cause the `TooMuchCheeseError` exceptions to remain unhandled, and will cause the program to terminate.

The previous solution, although elegant and efficient, has one important weakness. Due to the somewhat easygoing way of declaring the constructors, the new exceptions cannot be used as-is, without a full list of required arguments.

We'll remove this weakness by **setting the default values for all constructor parameters**. Take a look:

```
class PizzaError(Exception):
    def __init__(self, pizza='uknown', message=''):
        Exception.__init__(self, message)
        self.pizza = pizza

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='uknown', cheese='>100', message=''):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese

def makePizza(pizza, cheese):
        if pizza not in ['margherita', 'capricciosa', 'calzone']:
            raise PizzaError
        if cheese > 100:
            raise TooMuchCheeseError
        print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
        try:
            makePizza(pz, ch)
        except TooMuchCheeseError as tmce:
            print(tmce, ':', tmce.cheese)
        except PizzaError as pe:
            print(pe, ':', pe.pizza)
```

Now, if the circumstances permit, it is possible to use the class names alone.