Python Advanced Course

Part III

Stefano Alberto Russo

Outline

- Part I: Object Oriented Programming
  - What is OOP?
  - Logical Example
  - Attributes and methods
  - Why to use objects
  - Defining objects
- Part II: Improving your code
  - Extending objects
  - Lambdas
  - Comprehensions
  - Iterables
  - Properties
- Part III: Exceptions
  - What are exceptions?
  - Handling exceptions
  - Raising exceptions
  - Creating custom exceptions
- Part IV: logging and testing
  - The Python logging module
  - Basics about testing
  - The Python unit-testing module
  - Test-driven development

Exceptions

➔ *What are “exceptions”?*

Exceptions are the “errors” in Python.

Exceptions

➔ *What are “exceptions”?*

Exceptions are the “errors” in Python.

```
1
2   my_var = 'Ciao'
3   print(mia_variablie)
4
```

```
› python lezione5.py
Traceback (most recent call last):
  File "lezione5.py", line 3, in <module>
    print(mia_variablie)
NameError: name 'mia_variablie' is not defined
```

```
1
2   my_var = 'Ciao'
3   float(my_var)
4
```

```
› python lezione5.py
Traceback (most recent call last):
  File "lezione5.py", line 3, in <module>
    float(my_var)
ValueError: could not convert string to float: 'Ciao'
```

Exceptions

➔ *What are “exceptions”?*

Exceptions are the “errors” in Python.

And they are…

Exceptions

➔ *What are “exceptions”?*

Exceptions are the “errors” in Python.

And they are… ***objects***

Exceptions

➔ *What are “exceptions”?*

As objects, they leverage inheritance.

→ *They all extend the base class “Exception”*

Examples:

```
Exception
    ArithmeticError
        FloatingPointError
        ZeroDivisionError
    AttributeError
    SyntaxError
    NameError
    TypeError
    ValueError
```

Exceptions

➔ *Handling exceptions*

The try-except construct allows to handle exceptions. To “catch” them.

Exceptions

➔ *Handling exceptions*

The try-except construct allows to handle exceptions. To “catch” them.

```
my_var = 'Hello'

try:
    my_var = float(my_var)
except:
    print('Cannot convert my_var to float')
```

```
Cannot convert my_var to float
```

Exceptions

➔ *Handling exceptions*

I can react to errors (exceptions):

```
my_var = 'Hello'

try:
    my_var = float(my_var)
except:
    print('Cannot convert my_var to float')
    print('Will use the default value of 0.0')
    my_var = 0.0
```

```
Cannot convert my_var to float
Will use the default value of 0.0
```

Exceptions

➔ *Handling exceptions*

…and I can get the exception inside the “except” branch:

```
my_var = 'Hello'

try:
    my_var = float(my_var)
except Exception as e:
    print('Got error in converting: "{}"'.format(e))
    print('Will use the default value of 0.0')
    my_var = 0.0
```

```
Got error in converting: "could not convert string to float: 'Hello'"
Will use the default value of 0.0
```

Exceptions

➔ *Handling exceptions*

…and I can get the exception inside the “except” branch:

```
my_var = 'Hello'

try:
    my_var = float(my_var)
except Exception as e:
    print('Got error in converting: "{}"'.format(e))
    print('Will use the default value of 0.0')
    my_var = 0.0
```

To use the exception inside the “except”, I always have to specify which exception I want to handle, if I want to handle them all then I use the base class Exception.

```
Got error in converting: "could not convert string to float: 'Hello'"
Will use the default value of 0.0
```

Exceptions

➔ *Handling exceptions*

I can indeed handle multiple exceptions in a chain:

```
try:
    my_var = float(my_var)

except TypeError:
    print('Wrong type for float conversion: "{}"'.format(type(my_var)))

except ValueError:
    print('Wrong value for float conversion: "{}"'.format(my_var))

except Exception as e:
    print('Got generic error in converting: "{}"'.format(e))
```

Exceptions

➔ *Handling exceptions*

There are two more useful constructs the “else” and the “finally”:

```
my_file = open('/tmp/tmp.txt', 'w')

try:
    my_file.write(my_string)

except Exception as e:
    print('Got error in writing to file: "{}"'.format(e))

else:
    print('Success!')

finally:
    my_file.close()
```

Exceptions

➔ *Raising exceptions*

Exceptions can be raised by you as well:

```
def concat(a,b):
    if not isinstance(a, str):
        raise TypeError('First argument not of type string')
    if not isinstance(b, str):
        raise TypeError('Second argument not of type string')
    return a+b

print(concat('Hello', 67))
```

```
...
TypeError: First argument not of type string
```

Exceptions

➜ *Custom exceptions*

You can define custom exceptions by extending the base Exception class

```
class PortfolioError(Exception):
    pass

class EmptyPortfolioError(PortfolioError):
    pass
```

Exceptions

➔ *Raising exceptions*

And you can also catch and re-raise

```
def do_stuff(portfolio):
    try:
        first_portfolio_item = portfolio[0]
    except IndexError:
        raise EmptyPortfolioError('Portfolio is empty') from None

do_stuff([])
```

```
...
__main__.EmptyPortfolioError: Portfolio is empty
```

End of part III

➔ *Questions?*

Next: exercise 3

Exercise 3

Add proper exception handling and raising to the IncrementModel and FitIncrementModel.

Try to score up in the autograding!