Object Oriented Programming
Using
Python

Teemu Matilainen
teemu.matilainen@savonia.fi

Lecture 8#

- **You know how to use** comprehensions **within own classes**
- **You realize what exceptions are**
- **You realize how exceptions are used**
- **You are able to catch an exception**
- **You are able to throw an exception**
- **You are able to design an exception class of your own**

Exercise…

Create a MyShoppingList class and adjust the class so that it is **iterable** and can thus be used as follows:

```
class ShoppingListIter:
    def __init__(self):
        self.items = {}

    def add_item(self, item):
        self.items[item] = self.items.get(item, 0) + 1

    def __iter__(self):
        self.current_item = None
        self.item_iterator = iter(self.items.items())
        return self

    def __next__(self):
        try:
            self.current_item = next(self.item_iterator)
            return self.current_item
        except StopIteration:
            raise StopIteration
```

```
if __name__ == "__main__":
    shopping_list = MyShoppingList()
    shopping_list.add_item("Beer IV", 10)
    shopping_list.add_item("Chips", 5)
    shopping_list.add_item("Cheese", 1)

for product in shopping_list:
    print(f"{product[0]}: {product[1]} units")
```

iter()

- The iter() function is used to obtain an iterator from an object that implements the iteration protocol.

- If the object has a __iter__() method, the iter() function calls this method to get the iterator.
- If the object doesn't have a __iter__() method but has a __getitem__() method, iter() creates an iterator that accesses elements using integer indices.
- If neither method is present, iter() raises a TypeError.

next()

- The next() function is used to retrieve the next item from an iterator.
- It takes an iterator as its first argument and an optional default value as its second argument.
- If the iterator has more items, next() returns the next item; otherwise, it raises a StopIteration exception if no default value is provided.
- If a default value is provided, next() returns the default value when the iterator is exhausted.

Simple Iteration

```
my_iter = iter([1, 2, 3, 4])

value = next(my_iter, None)
print(value)
value = next(my_iter, None)
print(value)
value = next(my_iter, None)
print(value)
value = next(my_iter, None)
print(value)
```

Another more 'pythonic' approach to creating lists from existing ones is through the use of list ***comprehensions***. The concept involves condensing onto a single line both the operation applied to each item in the list and the assignment of the results to a new list.

Comprehensions With Own classes

```python
from book import Book
from antique_store import Antique_store

if __name__ == "__main__":
    book_1 = Book("Never ending storie 1", "123", 23)
    book_2 = Book("Never ending storie 2", "123", 34)
    book_3 = Book("Never ending storie 3", "123", 67)
    book_4 = Book("Never ending storie 4", "123", 789)

    books = [book_1, book_2, book_3, book_4]

    expensive_books = [book.name for book in books if book.price > 34]
    for b in expensive_books:
        print(b)
```

Comprehensions can prove valuable in handling or creating instances of custom classes, as demonstrated in the upcoming examples.

Comprehensions With Own classes

```python
from book import Book
from antique_store import Antique_store

if __name__ == "__main__":
    book_1 = Book("Never ending storie 1", "123", 23)
    book_2 = Book("Never ending storie 2", "123", 34)
    book_3 = Book("Never ending storie 3", "123", 67)
    book_4 = Book("Never ending storie 4", "123", 789)

    books = [book_1, book_2, book_3, book_4]

    #expensive_books = [book.name for book in books if book.price > 34]
    expensive_books = [book for book in books if book.price > 34]

    for b in expensive_books:
        print(b)
```

```
<book.Book object at 0x000001C80A06F830>
<book.Book object at 0x000001C80A06F770>
```

```python
expensive_books = [book.name for book in books if book.price > 34]
#expensive_books = [book for book in books if book.price > 34]

for b in expensive_books:
    print(b)
```

```
Never ending storie 3
Never ending storie 4
```

Comprehensions With Own classes

```python
from book import Book
from antique_store import Antique_store

if __name__ == "__main__":
    book_1 = Book("Never ending storie 1", "123", 23)
    book_2 = Book("Never ending storie 2", "123", 34)
    book_3 = Book("Never ending storie 3", "123", 67)
    book_4 = Book("Never ending storie 4", "123", 789)

    books = [book_1, book_2, book_3, book_4]

    #expensive_books = [book.name for book in books if book.price > 34]
    expensive_books = [book for book in books if book.price > 34]

    for b in expensive_books:
        print(b)

    a_store = Antique_store()
    a_store.add_book_to_shelf(book_1)
    a_store.add_book_to_shelf(book_2)
    a_store.add_book_to_shelf(book_3)
    a_store.add_book_to_shelf(book_4)

    # Create a list containing the names of all books
    book_names = [book.name for book in a_store]
    print(book_names)
```

```
['Never ending storie 1', 'Never ending storie 2', 'Never ending storie 3', 'Never ending storie 4']
```

Comprehension and class methods…

```
a_store = Antique_store()
a_store.add_book_to_shelf(book_1)
a_store.add_book_to_shelf(book_2)
a_store.add_book_to_shelf(book_3)
a_store.add_book_to_shelf(book_4)

# Create a list containing the names of all books
book_names = [book.name for book in a_store]
print(book_names)

# You can even invoke a class method that you have defined yourself:
book_summary = [book.summary() for book in a_store]
print(book_summary)
```

In reality, the expression used within the list comprehension statement can be any valid Python expression. You can even invoke a class method that you have defined yourself:

```python
def __iter__(self):
    self.index = 0
    self.keys = list(self.items.keys())
    return self

def __next__(self):
    if self.index < len(self.keys):
        index = self.index
        self.index += 1
        return self.keys[index], self.items[self.keys[index]]
    else:
        raise StopIteration
```

Comprehension for Shopping list

```python
list_of_list =  {item: quantity for item, quantity in shopping_list.items.items() if quantity >= 2 and quantity < 14}
print(list_of_list)
```

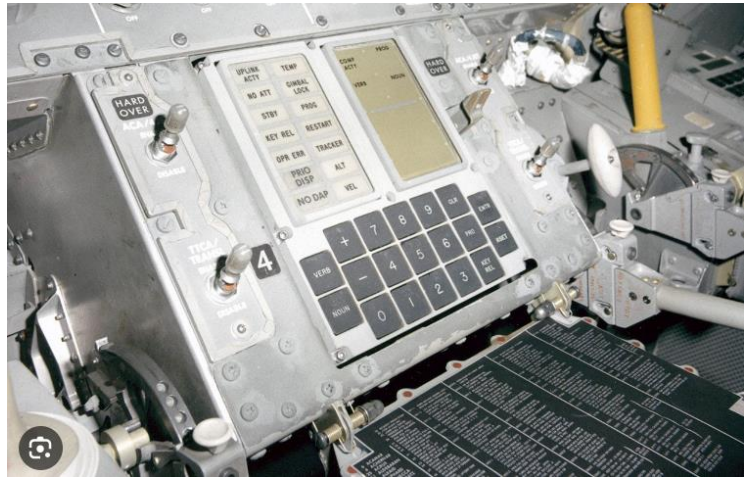
```
{'Beer IV': 10, 'Chips': 5}
```

Exception handling

Exception handling is a **crucial aspect** of Python programming. It allows you to gracefully handle errors and unexpected situations that may occur during the execution of your code. Python provides a mechanism to catch and deal with exceptions using the **try, except, finally, and else blocks**.

Exception handling

Its July 20, 1969, Apollo 11 mission was close to touchdown on the moon. **Neil Amstrong, Buzz Aldrin, and Michel Collins** checking last-minute preparation for landing. Over their, onboard computer something started flashing it was Error 1201 followed by error 1202. Human beings first landing on the moon is jeopardized by two computer errors…



Here's a basic example of exception handling in Python:

```
####################################
# Properly handled exception
####################################

try:
    # Code block that may raise an exception
    x = 10 / 0
except ZeroDivisionError:
    # Handle the specific exception
    print("Error: Division by zero!")

####################################
# Error depends on situation
####################################

i=int(input('Give me an integer: '))

####################################
# Error definately happens
####################################

print(10 / 0)
```

Exceptions?

What are exceptions?

- Exceptions are mechanism to handle runtime errors
- For example, user gives string ’one’ when asked an integer

⇒ Conversion throws an exception

⇒ If the exception is not handled correct, the software crashes

Catching Exceptions continues

You can also catch multiple exceptions and handle them

```
################################
# General
################################
try:
    a = int(input("Give me an integer value"))
except:
    print("It was not an integer value")

################################
# More accurate
################################
try:
    a = int(input("Give me and integer value: "))
except ValueError:
    print("It was not an integer value")
except:
    print("It was something else...")

################################
# Even more deep
################################
try:
    a = int(input("Give me and integer value: "))
except ValueError as value_error:
    print(f"It was not an integer value {value_error}")
except:
    print("It was something else...")
```

Catching Exceptions. Ain’t life wonderful?

```
##############################
# Ain't life wonderful?
##############################

try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    # Handle division by zero error
    print("Error: Division by zero!")
except ValueError:
    # Handle invalid input error
    print("Error: Invalid input! Please enter a valid number.")
```

You can also catch multiple exceptions and handle them differently…

Catching Exceptions. Finally we are getting somewhere… Or are we?

```
############################
# Finally we are getting somewhere...
############################

try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    # Handle division by zero error
    print("Error: Division by zero!")
finally:
    # Cleanup code (optional), always executed
    print("End of program.")
```

You can use the finally block to execute cleanup code regardless of whether an exception occurs!

Catching Exceptions. Is there anything else?

```
############################
# Is there anything else?
############################

try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    # Handle division by zero error
    print("Error: Division by zero!")
else:
    # Code to execute if no exceptions are raised
    print("Result:", result)
```

Else block can be used to execute code if no exceptions are raised:

Catching Exceptions. Else & Finally

```
############################
# Is there anything else?
############################

try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    # Handle division by zero error
    print("Error: Division by zero!")
else:
    # Code to execute if no exceptions are raised
    print("Result:", result)
finally:
    # Cleanup code (optional), always executed
    print("Cleanup code (optional), always executed")
```

Else + Finally

Catching Exceptions. All together!!

```
############################
# Is there anything else?
############################

try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ZeroDivisionError:
    # Handle division by zero error
    print("Error: Division by zero!")
except ValueError:
    # Handle invalid input error
    print("Error: Invalid input! Please enter a valid number.")
else:
    # Code to execute if no exceptions are raised
    print("Result:", result)
finally:
    # Cleanup code (optional), always executed
    print("Cleanup code (optional), always executed")
```

Raising Custom errors

To create your own custom exceptions in Python, you can subclass the built-in Exception class or any other built-in exception class. This allows you to define specific types of errors that are relevant to your application.

```
##############################
# Custom error handling
##############################

class CustomError(Exception):
    pass

def check_value(value):
    if value < 0:
        raise CustomError("Value cannot be negative.")

try:
    check_value(-5)
except CustomError as e:
    print("Error:", e)
```

Raising Custom errors continues…

You can create more specific custom exceptions by **subclassing other built-in exceptions** if your custom exception represents a specific type of error. For example, if your custom exception is related to file operations, you might subclass **IOError**.

```
############################
# Using buil-in-exceptions
############################
class FileOperationError(IOError):
    pass

def read_file(filename):
    try:
        with open(filename, 'r') as file:
            contents = file.read()
            # File is automatically closed when exiting the block
    except FileNotFoundError:
        raise FileOperationError(f"File '{filename}' not found.")

try:
    read_file("nonexistent_file.txt")
except FileOperationError as e:
    print("Error:", e)
```

The **with statement** ensures that the context management methods __enter__() and __exit__() of the object are properly called, **regardless** of whether the block of code inside the **with** statement raises an exception or not. This is commonly used for acquiring and releasing resources in a clean and predictable manner.

When working with files, using the with statement ensures that the file is properly closed after its use, even if an exception occurs!

Raising Custom errors continues…

- You can define your custom error class constructor attributes…

```
###################################
# Handling several custom errors
###################################

class ErrorValueNegative(Exception):
    def __init__(self, text, code):
        self.text = text
        self.code = code

class ErrorValuePositive(Exception):
    def __init__(self, text, code):
        self.text = text
        self.code = code

class Value:
    def __init__(self):
        pass

    def check_value(self, value: int):
        if value < 0:
            raise ErrorValueNegative("Value is negatine", 128)
        elif value > 0:
            raise ErrorValuePositive("Value is positive", 256)
        else:
            print("Value is OK = ", value)
```

```
if __name__ == "__main__":
    v = Value()

    try:
        v.check_value(int(input("Anna minulle luku nolla (0): ")))
    except ErrorValueNegative as e:
        print(e.text)
    except ErrorValuePositive as e:
        print(e.code)
    except ValueError as e:
        print(f"Error: ", e)
```

```python
if __name__ == "__main__":
    v = Value()

    try:
        v.check_value(int(input("Anna minulle luku n
    except ErrorValueNegative as e:
        print(e.text)
    except ErrorValuePositive as e:
        print(e.code)
    except ErrorNew as e:
        print(e.message + " " + str(e.code))
    except ValueError as e:
        print(f"Error: ", e)
```

Raising Custom errors…

- In Python, when you subclass a built-in class like Exception to create a custom exception class, it's generally a good practice to call the **__init__** method of the superclass (the class you're inheriting from) within your custom exception class's **__init__** method. This ensures that any initialization logic defined in the superclass is executed, which can be important for proper exception handling.

```python
# Handling several custom errors
#####################################

class ErrorNew(Exception):
    def __init__(self, *args: object):
        super().__init__(*args)
        self.message = args[0]
        self.code = args[1]

class ErrorValueNegative(Exception):
    def __init__(self, text, code):
        self.text = text
        self.code = code

class ErrorValuePositive(Exception):
    def __init__(self, text, code):
        self.text = text
        self.code = code

class Value:
    def __init__(self):
        pass

    def check_value(self, value: int):
        if value < 0:
            raise ErrorNew("Value is negatine
        elif value > 0:
            raise ErrorValuePositive("Value i
        else:
            print("Value is OK = ", value)
```

Assignment:

Create a program (class **AreaCalculator**) that calculates the area of geometric shapes.

The program should have “exception” classes: **ShapeError** as the base **exception** class, and two subclasses, **NegativeDimensionError** and **InvalidShapeError**.

- **ShapeError**: Base **exception** class representing errors related to shape calculations.
- **NegativeDimensionError**: Subclass of **ShapeError**, raised when the dimensions (length, width, etc.) provided to calculate the area are negative.
- **InvalidShapeError**: Subclass of **ShapeError**, raised when an invalid shape type is provided for calculation.

The program should include methods for calculating the **area of rectangles and circles**. Handle exceptions appropriately for negative dimensions and invalid shape types.

AreaCalculator has class methods:

- **rectangle_area**(self, length, width)
- **circle_area**(self, radius)
- **calculate_area**(self, shape_type, *args):

```python
if __name__ == "__main__":

    try:
        area1 = AreaCalculator()
        rectangle_area = area1.calculate_area('rectangle', 5, 3)
    except NegativeDimensionError as e:
        print("Caught NegativeDimensionError:", e.message, e.code)
    except InvalidShapeError as e:
        print("Caught InvalidShapeError:", e)
    else:
        print(rectangle_area)

    try:
        area2 = AreaCalculator()
        circle_area = area2.calculate_area('circle', 3)
    except NegativeDimensionError as e:
        print("Caught NegativeDimensionError:", e.message, e.code)
    except InvalidShapeError as e:
        print("Caught InvalidShapeError:", e)
    else:
        print(circle_area)
```

```
15
28.26
```

```python
 circle_area = area2.calculate_area('circle', -3)
 rectangle_area = area1.calculate_area('rectangle', 5, -3)
```

```
Caught NegativeDimensionError: Dimensions must be non-negative 12
Caught NegativeDimensionError: Radius must be non-negative 13
```

```python
circle_area = area2.calculate_area('dot', -3)
rectangle_area = area1.calculate_area('line', 5, -3)
```

```
Caught InvalidShapeError: ('Invalid shape type', 22)
Caught InvalidShapeError: ('Invalid shape type', 22)
```