Object oriented programming and exceptions in Python

Ivan Marković Matko Orsag Damjan Miklić

Automation and Robotics
Robot Programming and Simulation

2020

UNIVERSITY OF ZAGREB
Faculty of Electrical Engineering and Computing

Objects and Classes

- In simple terms, object = data + functions (methods)
- Support the concept of state
- Classes are "blueprints" of objects
- A natural and powerful way of thinking about problems
- Great for code modularity and reuse (if used right)
- Rule of thumb: Whenever you are tempted to use a global variable, use a class

Class example

Example

Implement 2D vectors which can be added.

```
class Vector2D:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def add(self, other):
        return Vector2D(self.x + other.x,
                        self.y + other.y)

if __name__ == '__main__':
    v1 = Vector2D(2,3)
    v2 = Vector2D(-1,4)
    print(v1.add(v2))
```

Magic methods

Magic methods

__*__ methods are magic, i.e., they allow us to overload operators and special functions, making our classes behave like built-in classes. A comprehensive guide is available here.

```
def __add__(self, other):
# ...
def __str__(self):
    return '({},{})'.format(self.x, self.y)
# ...
print(str(v1+v2))
```

Class syntax

The anatomy of a class:

- Definition begins with the `class` keyword
- Method definitions are indented, start with `def`
- Must have the `__init__` method (constructor)
- The constructor is called on object instantiation
- The first argument to all methods must be `self`
- `self` refers to the object instance (`this` in C++)
- `self` is passed implicitly on method calls
- `self` allows data sharing between methods
- Do not forget colons in class and method definitions :)

Using classes to structure your programs

Assignment

You are programming a robot which will perform food, snack and refreshment order deliveries in an office environment. In order to be able to load the appropriate amount of food on the robot and to plan an optimal delivery route (not in scope of this task), the orders have to be read and processed, and the total amount of each type of food has to be calculated, displayed on screen and written to a file.

Using classes to structure your programs

Assignment

You are programming a robot which will perform food, snack and refreshment order deliveries in an office environment. In order to be able to load the appropriate amount of food on the robot and to plan an optimal delivery route (not in scope of this task), the orders have to be read and processed, and the total amount of each type of food has to be calculated, displayed on screen and written to a file.

Assignment reworded

Implement order queue management for a snack delivery robot. The queue shall keep track of total amounts of ordered items, which must be updated every time an order is received or delivered. The queue shall also support reading single orders from a file and displaying the total amounts of ordered items.

Order queue management: a procedural implementation (1/2)

```
import sys

if __name__ == '__main__':

  totals = {}
  num_orders = 0
  with open(sys.argv[1],'r') as order_file:
    for name in order_file:
      num_orders += 1
      for idx in range(int(order_file.readline())):
        (item, quantity) = order_file.readline().split(': ')
        totals[item] = (float(qantity)
                        + totals.get(item,0.0))
```

Order queue management: a procedural implementation (2/2)

```
with open('totals_report.txt','w') as report_file:
  report_file.write('Processing {}\n'.format(sys.argv[1]))
  report_file.write('Number of orders: {}\n')
  report_file.write('Item totals:\n'.format(num_orders))
  for item in totals:
    report_file.write('\t{0}: {1}\n'.format(item,
                                            totals[item]))
```

Some issues with the procedural implementation

- Everything is implemented inside the `__main__` block
  - Poor readability
  - No modularity
  - Hard to re-use the code
- Reading from file and order queue management is intertwined
- Extending missing functionality (registering deliveries) will lead to further loop nesting

Object-oriented order queue: the __main__ block

```
if __name__ == '__main__':

  orders = OrderQueue()
  with open(sys.argv[1],'r') as order_file:
    for name in order_file:
      orders.receive(name.strip(),
                     orders.read_order(order_file))

  orders.deliver('Peter')
  with open('orders_report.txt','w') as report_file:
    report_file.write('Processed {}\n'.format(sys.argv[1]))
    report_file.write(str(orders))
  print(str(orders))
```

Object-oriented order queue: receive and deliver

```
class OrderQueue:
  def __init__(self):
    self.orders = {}
    self.pending = {}

  def receive(self, name, order):
    self.orders[name] = order
    for item in order:
      self.pending[item] = (order[item]
                            + self.pending.get(item,0.0))

  def deliver(self, name):
    for item in self.orders[name]:
      self.pending[item] -= self.orders[name][item]
    self.orders.pop(name)
```

Object-oriented order queue: read and print

```
def read_order(self, file):
  order = {}
  for idx in range(int(file.readline())):
    (item, quantity) = file.readline().split(': ')
    order[item] = float(quantity)
  return order

def __str__(self):
  str_ = 'Number of orders: {}\n'
  str_ += 'Item pending:\n'.format(len(self.orders.keys()))
  for item in self.pending:
    str_ += '\t{0}: {1}\n'.format(item, self.pending[item])
  return str_
```

Object-oriented order queue: testing

`orders.txt` input file format:

```
Peter
2
Ham sandwich: 1
Coke: 1.0
Alice
3
Veggie sandwich: 1
Dried apple slices: 1
Sparkling water: 0.33
...
```

```
$ ./process_orders.py orders.txt
```

Object and class gotchas

- Objects are mutable (i.e. they are passed around as references)!

OOP and general programming tips

- OOP[1] is all about code reuse
- Use pencil and paper before using the keyboard :)
- Write down a description of your program
  - Nouns are potential classes
  - Verbs are methods
- Break your program down into logical units
  - Functions
  - Classes
  - Modules
- Work incrementally:
  - Write a small chunk of code
  - Test it
  - Integrate
  - Repeat :)

[1] Object-Oriented Programming

Things do not always go according to plan!

An input file can be malformed, e.g.:

```
Peter
3 # Input error!
Ham sandwich: 1
Coke: 1.0
...
```

```
$ ./process_orders.py orders.txt
Traceback (most recent call last):
  File "./process_orders_queue.py", line 56, in <module>
    orders.receive(name.strip(),
                   orders.read_order(order_file))
  File "./process_orders_queue.py", line 41, in read_order
    (item, quantity) = file.readline().split(': ')
ValueError: not enough values to unpack (expected 2, got 1)
```

Handling exceptions (1/2)

```
def read_order(self, file):
  order = {}
  for idx in range(int(file.readline())):
    try:
      init_pos = file.tell()
      (item, quantity) = file.readline().split(': ')
      order[item] = float(quantity)
    except ValueError as ex:
      print('Error reading order!')
      print('Original error: {}'.format(str(ex)))
      file.seek(init_pos)
  return order
```

Handling exceptions (2/2)

We need to modify our file iteration loop to work with `seek` and `tell` methods:

```
# __main__ block
name = order_file.readline()
while name:
  orders.receive(name.strip(), orders.read_order(order_file))
  name = order_file.readline()
```

Exceptions

- Signaling and (hopefully :) handling irregular program conditions
- Allow jumping over arbitrary large chunks of code
- Unhandled exceptions propagate up the call stack
- Catching exceptions reduces the need of checking for status codes
- We can raise exceptions ourselves (don't overuse!)