KOLT Python

Error/Exception Handling, Object Oriented Programming

İpek Köprülülü

Monday 1st April, 2019

KOLT

Agenda

1. Recap

2. Error/Exception Handling

3. Object Oriented Programming

Sets

- **Unordered** sequence of **unique** elements.
- **Cannot** use **indexing/slicing**, **can** iterate with `for` loops.
- **Mutable**, `add(element)`, `remove(element)` methods.
- Python also has **immutable** sets: `frozenset`
- How to create sets? `my_set = {1, 2, 3, 4, 2}`
- How to create empty sets? `set()` ({ } is reserved for `dict`)
- Can compute set operations: **union**, **intersection**, **difference**, **symmetric difference**.

Sets

```
biology = {'Ashley', 'Patrick', 'Molly', 'Bob',
           'Mark', 'Matt'}
calculus = {'Becca', 'Shira', 'Alex', 'Molly', 'Bob', 'Steph'}
spanish = {'Matt', 'Mark', 'Bob', 'Alex', 'Steph', 'Julia', 'Andy'}
# intersection &
print(biology.intersection(calculus))  # => {'Molly', 'Bob'}
print(calculus & spanish)  # => {'Bob', 'Alex', 'Steph'}
# union |
print(biology.union(calculus))  # => all names except andy and julia
print(calculus | spanish | biology)  # => all names
# difference -
print((biology - calculus).intersection(spanish))  # => {'Mark', 'Matt'}
# symmetric_difference ^
print(biology.symmetric_difference(spanish))
# => {'Molly', 'Julia', 'Ashley', 'Alex', 'Steph', 'Andy', 'Patrick'}
```

Dictionaries

- Collection of **key**−**value** pairs.
- **Cannot** use **indexing**/**slicing**, **can** iterate with `for` loops.
- In general, they are not **ordered**.
- However, in Python 3.7 pairs are guaranteed to be in insertion order.
- In other words, we will get pairs in insertion order if we loop over the `dict`.
- How to create dictionaries? `{ }`/`dict()`: empty dictionary
- `d = {'one': 1, 'two': 2, 'three': 3, 'four': 4}`
- How to access values? `print(d['one'])` # ⇒ 1

Dictionaries

```
d = {'x': 1, 'y': 2, 'z': 3}
for key, value in d.items():
    print(f'value {value} is associated with key: {key}')

for key in d:
    print(f'value {d[key]} is associated with key: {key}')

# Add new pairs
d['a'] = 15
# Change value of key
d['x'] = 1
# Remove pairs
y_value = d.pop('y')
```

Files In Python

Access to a `file object` using **`open(filename, mode='r')`** function

- **`filename`**: File name including the **file extension**. Ex: 'data.txt'
- If you want to access/create a file outside of current **working directory**, you also need to include path. Ex: './FolderName/data.txt', 'C:/Users/AUYSAL16/Desktop/data.txt'
- **`mode`** denotes how the file will be used:
  - 'r': read mode, default
  - 'w': write mode, overrides the file contents if it already exists
  - 'x': create & write mode, similar to write mode gives error if file already exists
  - 'a': append mode, adds content to the end of file

File Methods

How to read file content?

- First open the file `f = open('my_file.txt')`
- `f.read()`: returns content of entire file as a string
- `f.readline(): returns a single line from file`
- **`for`** `line` **`in`** `f:` ⇒ Iterate over all lines
- `list(f)`/`f.readlines()`: read file lines to a list
- **Always** close the file when you are done: `f.close()`

File Methods

How to create/modify files?

- Open the file with a write enabled mode, e.g, `w, x, a`
- Ex: `f = open('my_file','w')`
- Use `f.write(string)` to write to file
- `file.write()` method **only** takes **`str`** values!
- Close the file when you are done.
- `f.close()`

Syntax Errors

What happens when you run a syntactically incorrect file?

```
for i in range(100)
print(i)
# SyntaxError: invalid syntax
```

```
while True:
print('Hello')
# IndentationError: expected an indented block
```

Easy to detect: Your code will not work :)

Runtime Exceptions

When a statement is **syntactically correct** does that mean we are safe?
`print(3/0), int('hello'), 'hello'[2] = 'a'`
How to be safe in these situations?

- Put `if` checks everywhere?
- Too much effort, and probably we cannot list every condition.
- Solution is `try-except-finally` blocks.

Try Except Blocks

```
try:
    <risky-statements >
    <risky-statements >
    <risky-statements >
    ...
except ValueError as valError:
    print('value error', valError)
except (RuntimeError, TypeError, NameError):
    print('One of the above errors, but not ValueError')
else:
    print('No errors')
finally:
    print('This always runs')
```

Try Except Blocks

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Objects

```
num = 2
type(num) # => <class 'int'>

string = "Hello"
type(string) # => <class 'str'>

my_list = [1, 2, 3]
type(my_list) # => <class 'list'>

my_tuple = (1, 2, 3)
type(my_tuple) # => <class 'tuple'>

dictionary = dict()
type(dictionary) # => <class 'dict'>
```

Objects

We can think objects as the objects in real life.

For example;
A phone is an object.
It has attributes such as its brand, its being On/Off etc.
It has methods such as turn off, volume up etc.

A person is an object.
It has attributes such as name, surname, age, education level etc.
It has methods such as increase age, change education level etc.

We will create classes to create new objects.

Classes

Use key **class**

```
class Person():
    name = "Jane"
    surname = "Doe"
    age = 25
    education = "University"
    def greet(self):
        print("Hello World!")
```

```
person = Person()
print(person.name) # => Jane
print(person.surname) # => Doe
print(person.age) # => 25
print(person.education) # => University
person.greet() # => Hello World!
```

Classes

However; if we create another person, that person's attributes will be same.

How can we define same type of objects with different attributes?

```
class Person():
    def __init__(self, name, surname, age, education):
        self.name = name
        self.surname = surname
        self.age = age
        self.education = education
    def greet(self):
        print("Hello World!")
```

__init__() method is constructor and called automatically when we create an object.

Classes

```
person = Person("Jane", "Doe", 25, "University")
print(person.name) # => Jane
print(person.surname) # => Doe
print(person.age) # => 25
print(person.education) # => University
person.greet() # => Hello World!

person = Person("John", "Doe", 22, "High School")
print(person.name) # => John
print(person.surname) # => Doe
print(person.age) # => 22
print(person.education) # => High School
person.greet() # => Hello World!
```

Classes

We can set some attributes as default.

```
class Person():
    def __init__(self, name = "NoInfo", age = 18, education = "NoInfo"):
        self.name = name
        self.age = age
        self.education = education

person = Person(age = 30)
print(person.name) # => NoInfo
print(person.age) # => 30
print(person.education) # => NoInfo
```

Methods in Classes

We use **`self`** as the first input.

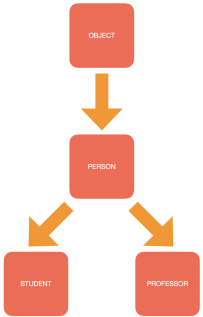
```
class Person():
    def __init__(self, name = "NoInfo", age = 18, languages = []):
        self.name = name
        self.age = age
        self.languages = languages

    def addLanguage(self, new_language):
        self.languages.append(new_language)

    def setName(self, name):
        self.name = name

    def increaseAge(self):
        self.age += 1

    def info(self):
        print("Name: {}\nAge: {}\nLanguages: {}".format(self.name, self.age, self.languages))
```

Methods in Classes

```
person = Person(age = 30)
print(person.name) # => NoInfo
print(person.age) # => 30
print(person.languages) # => []

person.addLanguage("English")
print(person.languages) # => ['English']

person.setName("John")
person.increaseAge()
person.increaseAge()

person.info() # => Name: John
              #    Age: 32
              #    Languages: ['English']
```

Inheritance

- All class objects inherit from object class.
- To implement same attributes of two different classes, we create a super class with their common traits.
- Subclass inherits superclass.
- We use `super()` while we mention the superclass.
- We can override the methods in superclass.



Inheritance

```
class Person():
    def __init__(self, name, surname, age):
        self.name = name
        self.surname = surname
        self.age = age
    def greet(self):
        print("Hello World!")
    def info(self):
        print("Name: {}\nSurname: {}\nAge: {}".format(self.name, self.surname, self.age))

class Student(Person):
    def __init__(self, name, surname, age, year, gpa):
        super().__init__(name, surname, age)
        self.year = year
        self.gpa = gpa
    def info(self):
        print("Name: {}\nSurname: {}\nAge: {}\nYear of Education: {}\nGPA: {}\n"
            .format(self.name, self.surname, self.age, self.year, self.gpa))
```

Inheritance

```
class Professor(Person):
    def __init__(self, name, surname, age, publication):
        super().__init__(name, surname, age)
        self.publication = publication
    def greet(self):
        print("Hello World, I'm a professor!")
    def info(self):
        print("Name: {}\nSurname: {}\nAge: {}\nNumber of Publication: {}\n"
            .format(self.name, self.surname, self.age, self.publication))
```

```
student = Student("Jane", "Doe", 24, 4, 3.50)
student.info()   #Name: Jane
                 #Surname: Doe
                 #Age: 24
                 #Year of Education: 4
                 #GPA: 3.5
student.greet()  #Hello World!
```

```
professor = Professor("Joe", "Roe", 43, 15)
professor.info()     #Name: Joe
                     #Surname: Roe
                     #Age: 43
                     #Number of Publication: 15
professor.greet()    #Hello World, I'm a professor!
```

Magic Methods

When we call the ones on the left, they automatically call the ones on the right.
There are many more magic methods.
str(x) => x.__str__()
x == y => x.__eq__(y)
x < y => x.__lt__(y)
x + y => x.__add__(y)
len(x) => x.__len__()
element in x => x.__contains__(element)

For example; when we use print() method on a person, it calls x.__str__() which is already defined for any object you'll create. However, Python cannot know what you mean by person1 < person2.

So, we should define the methods on the right to use the left methods.

Magic Methods

```
class Person():
    def __init__(self, name = "NoInfo", age = 0):
        self.name = name
        self.age = age

    def __lt__(self,other):
        return self.age < other.age

p1 = Person(age = 22)
p2 = Person(age = 25)

print(p1 < p2) # => True
```

Magic Methods

```python
class Person():
    def __init__(self, name = "NoInfo", age = 0):
        self.name = name
        self.age = age

    def __eq__(self,other):
        return self.name == other.name and self.age == other.age

p1 = Person("Jane", 22)
p2 = Person("Jane", 22)

print(p1 == p2) # => True
```