Energy Informatics

System Design — Data Modeling

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

08 Feb 2016

Who am I?

Who are you?

Data Modeling

- UML class diagrams
- Corresponding implementations
- Using Python as a vehicle

Jumping into Python

From the python.org website

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Jumping into Python

From the python.org website

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

What does that mean?

Jumping into Python

From the python.org website

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

What does that mean?

- **interpreted**: you can work interactively as with a pocket calculator

Jumping into Python

From the python.org website

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

What does that mean?

- **interpreted**: you can work interactively as with a pocket calculator
- **dynamic typing**: your programs just run, you don't have to fight with the system

UNI FREIBURG

Syntactic elements

- int(egers): `0`, `1`, `-1`, `42`, `-32768`, ...
- float(ing point numbers): `1.0`, `3.14159`, `.2288`, `-43.4` ...
- usual arithmetic operators: `+`, `-`, `*`, `/`

Python as a calculator
Numbers: int, float

Syntactic elements

- int(egers): `0`, `1`, `-1`, `42`, `-32768`, ...
- float(ing point numbers): `1.0`, `3.14159`, `.2288`, `-43.4` ...
- usual arithmetic operators: `+`, `-`, `*`, `/`

Talking to Python

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
5.0
>>> 8 / 5.0
1.6
```

Syntactic elements

- `"a string"`
- `'Monty Python\'s flying circus'`
- Operations: concatenation, indexing

Python as a calculator
Strings

Syntactic elements

- `"a string"`
- `'Monty Python\'s flying circus'`
- Operations: concatenation, indexing

Talking to Python

```
>>> 'Monty Python\'s flying circus'
"Monty Python's flying circus"
>>> 'Monty '  'Python'        # concatenation
'Monty Python'
>>> 'Monty' + ' ' + 'Python' # concatenation
'Monty Python'
>>> 'Monty Python'[4]         # index starts at 0
'y'
```

Syntactic elements

- variable names: `x`, `y`, `tissue`, `one_of`, ...
- assignment: `x = 1`, `y = 43.2`, `tissue = 'tempo'`

Python as a calculator
Variables

Syntactic elements

- variable names: `x`, `y`, `tissue`, `one_of`, ...
- assignment: `x = 1`, `y = 43.2`, `tissue = 'tempo'`

Talking to Python

```
>>> width = 42
>>> width
42
>>> width * 2
84
>>> height
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'height' is not defined
```

Syntactic elements

- empty list: `[]`
- enumerated lists:
  `[1, 3, 5, 7, 9]`, `['a', 'e', 'i', 'o', 'u']`
- operations: index and concatenation (like string)

Syntactic elements

- empty list: `[]`
- enumerated lists:
  `[1, 3, 5, 7, 9]`, `['a', 'e', 'i', 'o', 'u']`
- operations: index and concatenation (like string)

Talking to Python

```
>>> primes = [2, 3, 5, 7, 11]
>>> primes
[2, 3, 5, 7, 11]
>>> primes[3]
7
>>> primes + [13, 17, 19]
[2, 3, 5, 7, 11, 13, 17, 19]
```

Double the input

```
>>> def double(n):  # define function named 'double'
...     return 2*n  # return value of expression
...
>>> double(21)
42
>>> double("la")    # oops
'lala'
```

Temperature

Gauging the temperature of a drink

We want to gauge the temperature of (hot) coffee. The optimal drinking temperature is between 50 and 60 degrees centigrade.

Temperature

Gauging the temperature of a drink

We want to gauge the temperature of (hot) coffee. The optimal drinking temperature is between 50 and 60 degrees centigrade.

Python implementation

```
>>> def coffee_drinkable(temp):
...     return 50 <= temp <= 60
...     # returns a boolean, True or False
...
>>> coffee_drinkable(10)
False
>>> coffee_drinkable(100)
False
>>> coffee_drinkable(55)
True
```

More discerning temperature check

Coffee temperature

Given the temperature in a cup of coffee, return “too hot” if the temperature exceeds 60 degrees, “just right” if the temperature is between 50 and 60 degrees, and “too cold” if it is below 50.

More discerning temperature check

Coffee temperature

Given the temperature in a cup of coffee, return “too hot” if the temperature exceeds 60 degrees, “just right” if the temperature is between 50 and 60 degrees, and “too cold” if it is below 50.

Conditional

Solving this task requires a conditional.

Conditional for coffee judgment

```
>>> def coffee_judgment(temp):
...     if temp < 50:
...             return "too cold"
...     if temp < 60:
...             return "just right"
...     else:
...             return "too hot"
...
>>> coffee_judgment(45)
'too cold'
>>> coffee_judgment(55)
'just right'
>>> coffee_judgment(65)
'too hot'
```

Task: solve $ax^2 + bx + c = 0$ using the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Implementation of quadratic formula

```
>>> import math
>>> def midnight(a, b, c):
...     return (-b + math.sqrt(b*b - 4*a*c))/2/a
...
>>> midnight(1,0,-1)
1.0
```

Looks good! 1.0 is a root of $x^2 - 1 = (x + 1)(x - 1)$

- but what about the other root -1.0 of $x^2 - 1$?

- but what about the other root -1.0 of $x^2 - 1$?
- we could return a list of roots!

- but what about the other root -1.0 of $x^2 - 1$?
- we could return a list of roots!

Revised implementation of quadratic formula

```
>>> def midnight2(a, b, c):
...     d = b*b - 4*a*c
...     return [(-b + math.sqrt(d))/2/a,
...             (-b - math.sqrt(d))/2/a]
...
>>> midnight2(1,0,-1)
[1.0, -1.0]
```

- but what about the other root -1.0 of $x^2 - 1$?
- we could return a list of roots!

Revised implementation of quadratic formula

```
>>> def midnight2(a, b, c):
...     d = b*b - 4*a*c
...     return [(-b + math.sqrt(d))/2/a,
...             (-b - math.sqrt(d))/2/a]
...
>>> midnight2(1,0,-1)
[1.0, -1.0]
```

- Ok, got both now ... are we done?

Two further tests: $x^2 + 2x + 1 = 0$ and $x^2 + 1 = 0$

Testing the implementation

```
>>> midnight2(1,2,1)
[-1.0, -1.0]
>>> # unsatisfactory. should return one value

>>> midnight2(1,0,1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in midnight2
ValueError: math domain error
>>> # oops! this one has no real roots!
```

Facts from mathematics

Consider equation E:

$$ax^2 + bx + c = 0$$

Let $d = b^2 - 4ac$

- E has two distinct real solutions if $d > 0$
- E has one real solution if $d = 0$
- E has no real solutions if $d < 0$

We need to model this case distinction in the `midnight` function using a conditional `if`, `else`.

Case distinction: if-else

Final implementation of quadratic formula

```
>>> def midnight3(a, b, c):
...     d = b*b - 4*a*c
...     if d < 0:
...         return []
...     elif d == 0:
...         return [-b/2/a]
...     else:
...         return [(-b + math.sqrt(d))/2/a,
...                 (-b - math.sqrt(d))/2/a]
...
>>> midnight3(1,0,-1)
[1.0, -1.0]
>>> midnight3(1,2,1)
[-1]
>>> midnight3(1,0,1)
[]
```

Task

Write a function `check_first` that takes a string and a character and checks whether it matches the first character of the string.

Task

Write a function `check_first` that takes a string and a character and checks whether it matches the first character of the string.

Solution

```
>>> def check_first(str, ch):
...     return str[0] == ch
...
>>> check_first('Larynx', 'L')
True
>>> check_first('atama', 'x')
False
>>> check_first([2,3,5], 2) # works for lists!
True
```

Task

Write a function `count` that takes a string and a character and counts how often it occurs in the string.

Functions
Count occurrences of letter

Task

Write a function count that takes a string and a character and counts how often it occurs in the string.

Solution

```
>>> def count_element(str, ch):
...     count = 0
...     for c in str:
...         if c == ch:
...             count = count+1
...     return count
...
>>> count_element('atama', 'a')
3
>>> count_element('atama', 'x')
0
```

For loops

```
for c in str:
    body
    :
```

- c must be a variable name
- str stands for a list or a string (for example)
- body and subsequent lines aligned with it are executed once for each element (character) of str
- the variable c contains the current character

Dictionaries

Special datatype in scripting languages

- A dictionary stores an association between **keys** and **values**.
- Strings and numbers can serve as keys (among others).

Dictionaries

Special datatype in scripting languages

- A dictionary stores an association between **keys** and **values**.
- Strings and numbers can serve as keys (among others).

Talking to Python

```
>>> tel = { "gl": 8121, "cs": 8181 }
>>> tel["pt"] = 8051
>>> tel['cs']
8181
>>> del tel['cs']
>>> tel
{'gl': 8121, 'pt': 8051}
>>> tel['cs']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cs'
```

Applcation of dictionaries

Task

Count all letters in a string.

Applcation of dictionaries

Task

Count all letters in a string.

Python source

```
def count_all_letters(s):
    d = dict(); # empty dictionary
    for c in s:
        d[c] = d[c] + 1 if c in d else 1
    return d
```

Applcation of dictionaries

Task

Count all letters in a string.

Python source

```
def count_all_letters(s):
    d = dict(); # empty dictionary
    for c in s:
        d[c] = d[c] + 1 if c in d else 1
    return d
```

Example uses

```
>>> count_all_letters("atama")
{'a': 3, 'm': 1, 't': 1}
>>> count_all_letters("einnegermitgazellezagtimregennie'
{'a': 2, 'e': 8, 'g': 4, 'i': 4, 'm': 2, 'l': 2, 'n': 4,
```

Classes and Class Diagrams

Simple Classes

A **class** is similar to an entity. It describes compound data that consists of subsidiary data (called **attributes**) collected in an **instance** of the class. Additionally, it can describe **operations** on that data (later).

Example for simple class: Tea

Class description for Tea

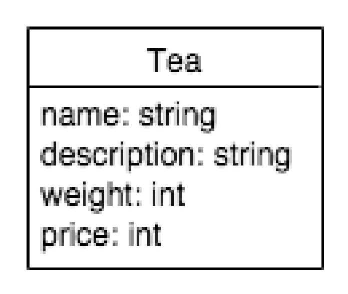
A tea shop describes a particular brand of **tea** in stock by its **name**; a **description** of its color, flavor, etc; the **weight** in stock (in g); and its **price** in cent per kg.

Example for simple class: Tea

Class description for Tea

A tea shop describes a particular brand of **tea** in stock by its **name**; a **description** of its color, flavor, etc; the **weight** in stock (in g); and its **price** in cent per kg.

Class diagram for Tea



Simple Classes in Python

A class diagram can be mapped line-by-line to (Python) code.

Class declaration

```
>>> class Tea:
...     def __init__(self,name,desc,wgt,price):
...         self.name = name
...         self.description = desc
...         self.weight = wgt
...         self.price = price
...
```

- `__init__` is a function that is called, when a new `Tea` instance is created. The `self` parameter is the new instance, `name`, `desc`, `wgt`, and `price` are used to initialize the respective attributes as shown.

Using simple classes

Creating and examining tea

```
>>> earl_grey = Tea("Earl Grey",
                    "Flavored black tea",
                    10000, 4335)
>>> earl_grey
<__main__.Tea instance at 0x1051dd950>
>>> earl_grey.name     # get name attribute
'Earl Grey'
>>> earl_grey.price    # get price attribute
4335
```

- `Tea()` creates a new Tea instance and calls its `__init__` method
- Access attributes using *`instance.attribute`*

Simple class with operation

Extended class description for Tea

A tea shop describes a particular brand of **tea** in stock by its **name**; a **description** of its color, flavor, etc; the **weight** in stock (in g); and its **price** in cent per kg. The shop wants to determine the stock value. It also wants to be able to print an inventory line.

Simple class with operation

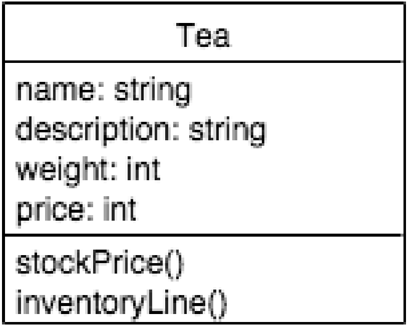
Extended class description for Tea

A tea shop describes a particular brand of **tea** in stock by its **name**; a **description** of its color, flavor, etc; the **weight** in stock (in g); and its **price** in cent per kg. The shop wants to determine the stock value. It also wants to be able to print an inventory line.

Two operations

- `stockPrice()`: no parameters, return total value of the tea brand in stock
- `inventoryLine()`: no parameters, return a string for printing the tea as an inventory item

Revised class diagram



- The implementation of `stockPrice` and `inventoryLine` belongs to the class declaration.
- Their first parameter is `self` and they can access all attributes.

Revised class declaration

```
class Tea:
    # __init__ omitted (same as before)
    def stockPrice(self):
        return self.weight * self.price / 1000
    def inventoryLine(self):
        return (self.name + '. ' +
                self.description + '. ' +
                str(self.weight) + 'g. ' +
                str(self.price) + ' c/kg.')
```

Remarks

- `str()` converts a number to a string

Meter Readings

Reading

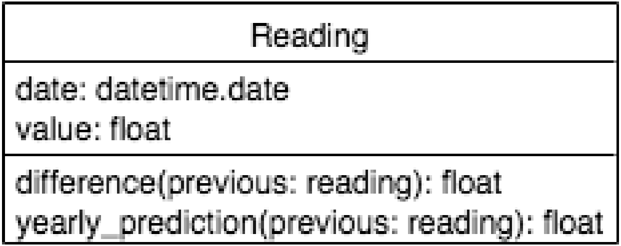
A reading of a metering device consists of a **reading date** and a **reading value**.

Meter Readings

Reading

A reading of a metering device consists of a **reading date** and a **reading value**.

Class diagram



Meter Readings implemented

Explanation

- `datetime` is a **module** that contains utilities for manipulating dates
- made available using
  `import datetime`

Meter Readings implemented

Implementation

```
import datetime

class Reading:
    def __init__(self, date, value):
        self.date = date     # datetime.date
        self.value = value   # float
    def difference(self, previous):
        return self.value - previous.value
    def yearly_prediction(self, previous):
        value_diff = self.value - previous.value
        date_diff = self.date - previous.date
        factor = 365.25 / date_diff.days
        return value_diff * factor
```

Compound Classes

Household

A household has an allocated amount of space (in square meters) and a number of occupants. Furthermore, a household has meter readings for several dates in the past.

Compound Classes

Household

A household has an allocated amount of space (in square meters) and a number of occupants. Furthermore, a household has meter readings for several dates in the past.

Class diagram

Association: Household — Reading

- The connection between `Household` and `Reading` in the class diagram is an **association**.
- It comes with a direction (arrow) that indicates the direction in which it can be travesed.
- We (choose to) represent the association with a list of readings stored in the Household instance.
- Requires a “housekeeping” method to add new readings.

Implementing Household

```
class Household:
    def __init__(self, space, occupants):
        self.space     = space
        self.occupants = occupants
        self.readings  = []
    def add_reading(self, reading):
        self.readings  = [reading] + self.readings
```

Further Household Methods

Requirements

For a household, we want to be able to determine the number of readings taken. If there are multiple readings, we want to give a statistical yearly prediction.

Implementation

```
class Household:        # __init__ ... as before
    def nr_readings(self):
        return len(self.readings)
    def yearly_average(self):
        if len(self.readings) < 2:
            return None  # more than one reading
        first_reading = self.readings[-1]
        last_reading  = self.readings[0]
        return last_reading.yearly_prediction(
                                    first_reading)
```

End Part I