Energy Informatics

System Design — Data Modeling

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

14 Feb 2017

Loops

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` from left to right
- variable `c` contains the current character

More generally

The same code works for other sequences

For example, for arrays

```
>>> count_element([1,2,3,2,1,2], 2)
3
>>> count_element([1,2,3,2,1,2], 4)
0
```

Summing the contents of an array

Task

Write a function `average` that takes an array with numbers and computes its arithmetic average.

Summing the contents of an array

Task

Write a function average that takes an array with numbers and computes its arithmetic average.

Solution

```
>>> def mysum(seq):  # predefined as sum
...     s = 0
...     for x in seq:
...         s += x
...     return s
...
>>> def average(seq):
...     return sum(seq) / len(seq)
```

Summing the contents of an array

Task

Write a function average that takes an array with numbers and computes its arithmetic average.

Solution

```
>>> def mysum(seq):  # predefined as sum
...     s = 0
...     for x in seq:
...         s += x
...     return s
...
>>> def average(seq):
...     return sum(seq) / len(seq)
```

Ok?

Missing a special case

What if `len(seq)==0`?

```
>>> average([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in average
ZeroDivisionError: division by zero
```

Missing a special case

What if `len(seq)==0`?

```
>>> average([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in average
ZeroDivisionError: division by zero
```

Safeguard such situations

Let's define that the average of an empty list is 0. This is an arbitrary choice, which is problem dependent.

```
def average0(seq):
    if len(seq) > 0:
        return sum(seq) / len(seq)
    else:
        return 0
```

Range

`range(b)` enumerates the elements of the list `[0, 1,..., b-1]`

```
>>> for i in range(10):
...     print ("{:5}{:5}".format(i, i*i))
...
    0    0
    1    1
    2    4
    3    9
    4   16
    5   25
    6   36
    7   49
    8   64
    9   81
```

How can we generalize?

How many positions are needed to print n^2?

```
def positions(n):
    return math.floor(2 + math.log10(n*n))
```

How can we generalize?

How many positions are needed to print n^2?

```
def positions(n):
    return math.floor(2 + math.log10(n*n))
```

How to create the format string?

```
    p = positions(n)
    f = "{{:{0}}}{{:{0}}}".format(p)
```

How can we generalize?

How many positions are needed to print n^2?

```
def positions(n):
    return math.floor(2 + math.log10(n*n))
```

How to create the format string?

```
    p = positions(n)
    f = "{{:{0}}}{{:{0}}}".format(p)
```

Putting it all together

```
def squares(n):
    p = positions(n)
    f = "{{:{0}}}{{:{0}}}".format(p)
    for i in range(n):
        print(f.format(i, i*i))
```

More about ranges

`range(b)`

Enumerates `0, 1, ..., b-1`

More about ranges

`range(b)`

Enumerates `0, 1, ..., b-1`

`range(a,b)`

Enumerates `a, a+1, ..., b-1`
Nothing if `a>=b`

More about ranges

`range(b)`

Enumerates `0, 1, ..., b-1`

`range(a,b)`

Enumerates `a, a+1, ..., b-1`
Nothing if `a>=b`

`range(a,b,s)` for `s>0`

Enumerates `a, a+s, ..., a+n*s`
where `n` is chosen maximal such that `a+n*s<b`
that is, if `s>0`, `n<(b-a)/s` which means
`n = math.floor ((b-a)/s)`

More about ranges

`range(b)`

Enumerates `0, 1, ..., b-1`

`range(a,b)`

Enumerates `a, a+1, ..., b-1`
Nothing if `a>=b`

`range(a,b,s)` for `s>0`

Enumerates `a, a+s, ..., a+n*s`
where `n` is chosen maximal such that `a+n*s<b`
that is, if `s>0`, `n<(b-a)/s` which means
`n = math.floor ((b-a)/s)`

There is also a story for `s<0` ...

Checking a range

Printing does not help

```
>>> r = range(10)
>>> print(r)
range(0, 10)
```

Checking a range

Printing does not help

```
>>> r = range(10)
>>> print(r)
range(0, 10)
```

Converting to a list

```
>>> [i for i in r] # a list comprehension
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions

Examples of *list comprehensions*

```
>>> S = [x*x for x in range(10)]
>>> V = [2**i for i in range(9)]
>>> M = [x for x in S if x % 2 == 0]
>>> print (S); print (V); print (M)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 2, 4, 8, 16, 32, 64, 128, 256]
[0, 4, 16, 36, 64]
```

Computing with lists

Dot product

```
def dotproduct(a, b):
    r = 0
    for i in range(min(len(a), len(b))):
        r += a[i]*b[i]
    return r
```

Computing with lists

Dot product

```
def dotproduct(a, b):
    r = 0
    for i in range(min(len(a), len(b))):
        r += a[i]*b[i]
    return r
```

Alternative approach: list comprehension

```
sum ([a[i]*b[i]
     for i in range(min(len(a), len(b))])
```

Compute the longest word in a text

Task

Given a text (as a string) find the longest word in it.

Compute the longest word in a text

Task

Given a text (as a string) find the longest word in it.

Subtasks

1. find all words in a string (result: a list)
2. find the longest word in a list

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'
```

Application of dictionaries

Task

Count the number of occurrences of all letters in a string.

Application of dictionaries

Task

Count the number of occurrences of 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
```

Application of dictionaries

Task

Count the number of occurrences of 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,
```

Alternative implementation of countletters

```
def count_letters(s):
    d = {}
    for c in s:
        if c in d:
            d[c] = d[c] + 1
        else:
            d[c] = 1
    return d
```

- Before using `d[c]`, we need to check whether `c in d`, that is, whether `c` is a defined key in dictionary `d`

Generalizing

N.B.

The code for `count_all_letters` does not depend on strings or letters. It can be used generally to collect the count of all different elements of a sequence. Examples for sequences:

- Strings — letter count
- List of numbers
- List of words — word count

Classes and Objects

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

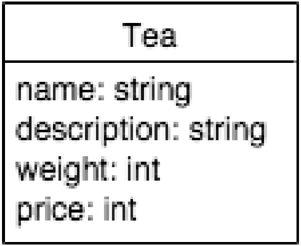
A tea shop describes a particular brand of **tea** in stock by its **name**; a **description** of its color, flavor, etc; the **weight** in stock (in g); and its **price** in cent per kg.

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

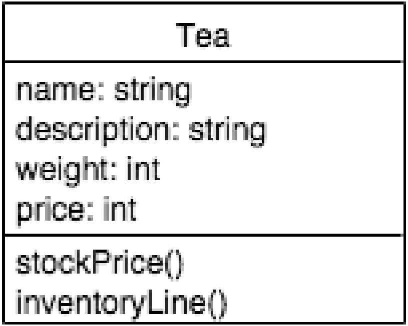
Extended class description for Tea

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

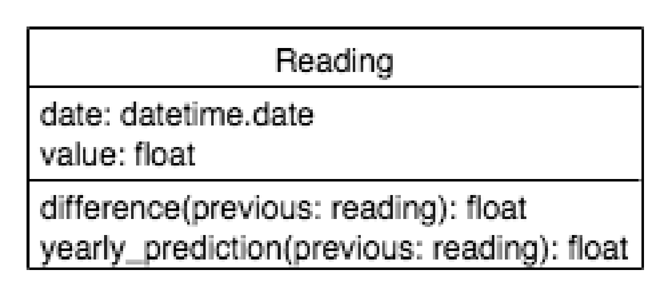
A reading of a metering device consists of a **reading date** and a **reading value**.

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

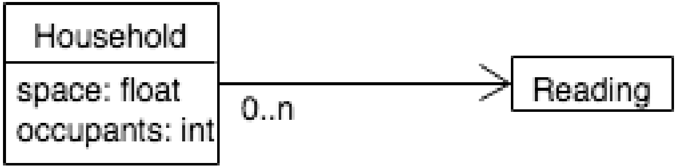
A household has an allocated amount of space (in square meters) and a number of occupants. Furthermore, a household has meter readings for several dates in the past.

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)
```

Data Modeling II

Data Modeling II

- Union
- Abstraction
- Inheritance

Union of classes

Task

A drawing program wants to manage different geometric shapes in a coordinate system. Initially, there are three kinds of figures:

- squares with reference point upper left and given side length
- circles with reference point in the middle and a given radius
- points that just consist of the reference point

Union of classes

Task

A drawing program wants to manage different geometric shapes in a coordinate system. Initially, there are three kinds of figures:

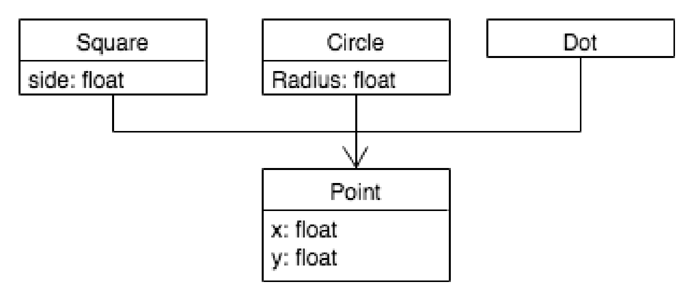
- squares with reference point upper left and given side length
- circles with reference point in the middle and a given radius
- points that just consist of the reference point

Approach

- Each kind of figure can be represented by a compound class. The reference point is a separate Point object.
- In many languages, they could not be used together, but no problem in Python

Union of classes

UML class diagram



Python implementation

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Square:
    def __init__(self, ref, side):
        self.ref = ref
        self.side = side
```

- and so on

Functionality for shapes

Task

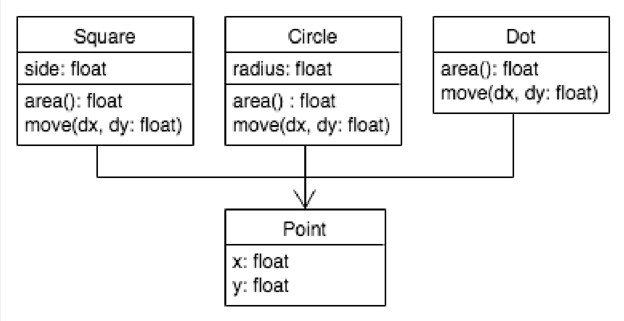
For each shape, we want to be able to compute the area and we want to move it around.

Functionality for shapes

Task

For each shape, we want to be able to compute the area and we want to move it around.

UML diagram



Python implementation

Square

```
def area(self):
    return self.side * self.side
def move(self, dx, dy):
    self.ref.move (dx, dy)
```

Python implementation

Square

```
def area(self):
    return self.side * self.side
def move(self, dx, dy):
    self.ref.move (dx, dy)
```

Circle

```
def area(self):
    return 2 * math.pi * self.radius
def move(self, dx, dy):
    self.ref.move (dx, dy)
```

Python implementation

Square

```
def area(self):
    return self.side * self.side
def move(self, dx, dy):
    self.ref.move (dx, dy)
```

Circle

```
def area(self):
    return 2 * math.pi * self.radius
def move(self, dx, dy):
    self.ref.move (dx, dy)
```

Dot . . .

Python implementation II

- All implementations assume a `move` method in `Point`.

Point

```
def move (self, dx, dy):
    self.x += dx
    self.y += dy
```

Python implementation II

- All implementations assume a `move` method in `Point`.

Point

```
def move (self, dx, dy):
    self.x += dx
    self.y += dy
```

Observation

- the move methods in `Square`, `Circle`, and `Dot` are all identical
- it would be nice to be able to advertise that all shape classes have methods `move` and `area`.

Abstraction

Abstraction in programming

- identify programming patterns
  repeated program fragments with similar semantics
- generalization
  replace specific parts by variables
- extraction
  give a name to the thus generalized program fragment
  invoke in the original places

Abstraction

Abstraction in programming

- identify programming patterns
  repeated program fragments with similar semantics
- generalization
  replace specific parts by variables
- extraction
  give a name to the thus generalized program fragment
  invoke in the original places

What does that mean?

- generally avoid duplication
- look for similarities
- try to solve each problem only once

Similarity among classes

Goal

- identify similar field and method declarations

Similarity among classes

Goal

- identify similar field and method declarations
- example: `Square.move`, `Circle.move`, `Dot.move`

Similarity among classes

Goal

- identify similar field and method declarations
- example: `Square.move`, `Circle.move`, `Dot.move`
- approach: introduce common **super class** `Shape`

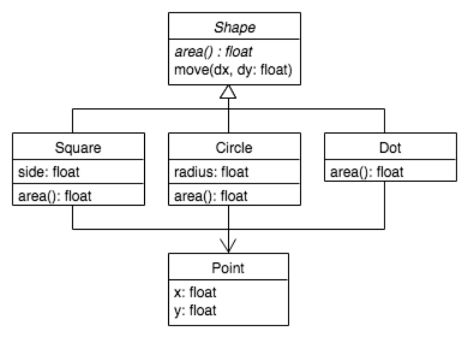
Similarity among classes

Goal

- identify similar field and method declarations
- example: `Square.move`, `Circle.move`, `Dot.move`
- approach: introduce common **super class** `Shape`
- indicated by arrow with open triangle head

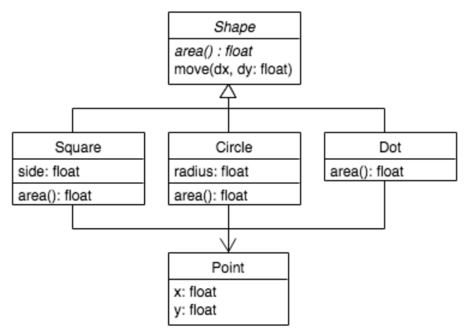
Inheritance

UML diagram: shapes



Inheritance

UML diagram: shapes



Italics indicate abstract items

- *Shape* is an **abstract class**: no instances
- *Shape.area()* is an **abstract method**: no implementation

Inheritance in Python

Super class Shape

```
class Shape:
    def __init__(self, ref):
        self.ref = ref
    def move(self, dx, dy):
        self.ref.move(dx, dy)
    def area(self):
        return 0
```

- it's not easily possible to define proper abstract classes in Python (you can create `Shape` instances)
- it's not possible to define abstract methods in Python; the way to do it would be to drop the definition of `area()`

Subclasses in Python

Square

```
class Square (Shape):
    def __init__ (self, ref, side):
        Shape.__init__(self, ref)
        self.side = side
    def area(self):
        return self.side * self.side
```

Subclasses in Python

Square

```
class Square (Shape):
    def __init__ (self, ref, side):
        Shape.__init__(self, ref)
        self.side = side
    def area(self):
        return self.side * self.side
```

Notes

- call `__init__` method of the super class `Shape`
- no need to define `move()`,
  its definition is **inherited** from `Shape`
- **override** `Shape`'s definition of `area()`

Exploiting inheritance

Weather data

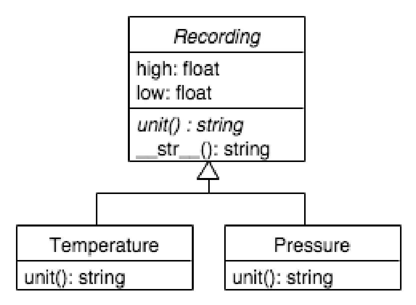
We want to keep track of various recordings of weather data all comprising of a high and a low reading. Two examples are temperature and pressure readings. All should be printable.

Exploiting inheritance

Weather data

We want to keep track of various recordings of weather data all comprising of a high and a low reading. Two examples are temperature and pressure readings. All should be printable.

Consider this class diagram



Implementing weather data

Printable

If a Python object has a method `__str__`, then that method is used to convert the object to a string.

Implementing weather data

Printable

If a Python object has a method `__str__`, then that method is used to convert the object to a string.

Printable Recording

```
class Recording:
    def __init__(self, low, high):
        self.low = low
        self.high = high
    def __str__(self):
        return (str (self.low) + '␣-␣' +
                str (self.high) + '␣' +
                self.unit())
```

Template Method

Printable Temperature recording

Temperature/Pressure can inherit printing from `Recording`, but it has to define the `unit()` method to make printing work!

Template Method

Printable Temperature recording

Temperature/Pressure can inherit printing from `Recording`, but it has to define the `unit()` method to make printing work!

Implementing concrete recordings

```
class Temperature (Recording):
    def unit():
        return "degrees"

class Pressure (Recording):
    def unit():
        return "hPa"
```

End Part II