Introduction to programming
Lecture 5

UNIVERSITY OF
GOTHENBURG

Richard Johansson

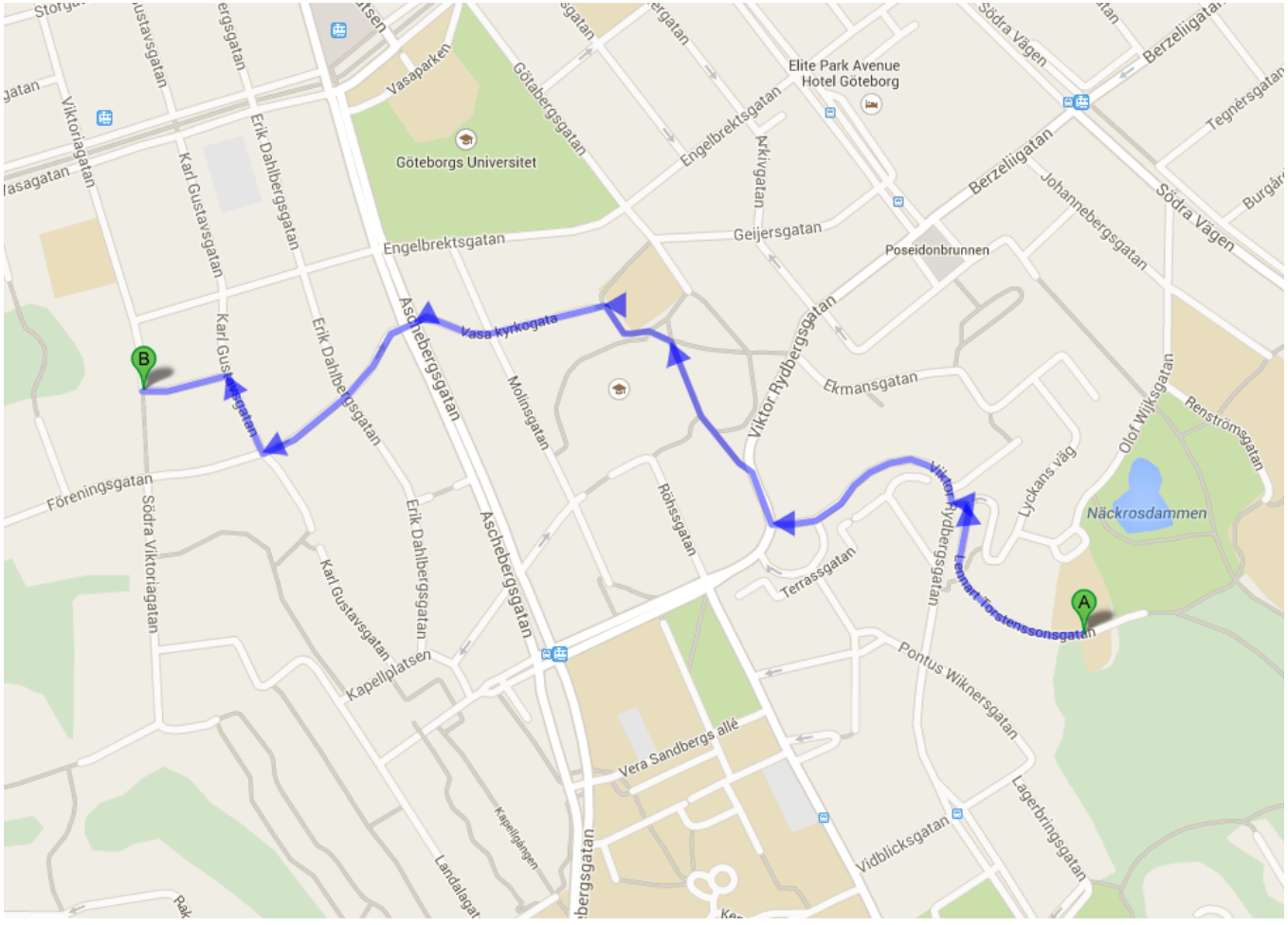
September 29, 2015

the exam

- location: Viktoriagatan 30
- time: **October 19, 9:00–12:00**, make sure to be on time!
- bring a valid ID document
- **you will need to register using GUL at least a week before**
  - select "Ladok Services", then "Examination Sign-up"
  - if you have trouble registering, ask the administrators at FLoV
- in lecture 7, we will go through an old exam

`http://www.styrdokument.adm.gu.se/digitalAssets/1344/1344035_rules-for-examinations.pdf`

Viktoriagatan 30



UNIVERSITY OF GOTHENBURG

overview of today's lecture

- recap last lecture
- more about repetition: `while`, `continue`, `break`, recursion
- higher-order functions: functions using functions
- introduction to user-defined types: classes

overview

recap files, dictionaries, sorting

while loops and recursion

higher-order functions

classes and objects

opening, reading, writing, ...

```
def read_a_file(filename):
    with open(filename) as f:
        content = f.read()
        return content

def write_some_text(filename, text):
    with open(filename, "w") as f:
        print(text, file=f)
```

dictionaries

```
tag_dict = { 'dog': 'noun',
             'in': 'preposition',
             'nice': 'adjective' }

tag_dict['who'] = 'relative pronoun'
tag_dict['little'] = 'adjective'

for word in ['nice', 'and', 'little']:
    if word in tag_dict:
        tag = tag_dict[word]
        print("The part-of-speech tag of %s is %s" % (word, tag)
    else:
        print("%s is not listed" % word)

for word in tag_dict:
    print("%s -> %s" % (word, tag_dict[word]))
```

example: counting words

```
from nltk.tokenize import word_tokenize, sent_tokenize

def compute_word_frequencies(filename):
    frequencies = {}
    with open(filename) as f:
        content = f.read()
        for sen in sent_tokenize(content):
            for word in word_tokenize(sen):
                if word in frequencies:
                    frequencies[word] += 1
                else:
                    frequencies[word] = 1
    return frequencies

freqs = compute_word_frequencies("test.txt")
print(freqs["the"])
```

sorting

- either `thelist.sort()` or `sorted(thelist)`
  - the first alternative sorts the list in place, while the second creates a new list
  - the second alternative can be used on any collection
- `sorted(list_of_strings, key=len)`
  - `sort` and `sorted` are higher-order functions: they use another function as input (`key`)
  - if no `key` is given, we will use the natural order (<)
- `sorted(list_of_strings, key=len, reverse=True)`

tuples

- **tuples** are fixed-size lists that cannot be changed
  - a tuple with 2 items is called a *pair*
  - a tuple with 3 items is called a *triple*
  - a tuple with *n* items is called an *n-tuple*
- tuples are more efficient than normal lists
- they are written with round brackets: `t = (3, "xyz")`
- like lists, we access its item using square brackets: `t[0]`

returning multiple values

- tuples are often used to return multiple values from a function

```
def get_first_and_last_name(full_name):
    ...
    return (first_name, last_name)

p = get_first_and_last_name("John Smith")
first = p[0]
last = p[1]
print(first)
```

- if a function returns multiple values, we can get them nicely if we use tuple **unpacking**

```
first, last = get_first_and_last_name("John Smith")
print(first)
```

ordering and sorting tuples

- useful fact about tuples: they can be compared
  - will compare by first item, then by second item, ...
- ...so if we have a list of tuples, it can be sorted

```
pairs1 = [ (6, "xyz"), (3, "ghi"), (5, "abc") ]
pairs2 = [ ("xyz", 6), ("ghi", 3), ("abc", 5) ]

print(sorted(pairs1))
print(sorted(pairs2))
```

key–value tuples from dictionaries

- if we have a dictionary `d`, the method `d.items()` gives a list of key–value pairs

```
email_dict = { "Richard":"richard.johansson@svenska.gu.se",
               "Johan":"johan.roxendal@svenska.gu.se",
               "Simon":"simon.dobnik@ling.gu.se" }

for name, email in email_dict.items():
    print("Name: %s, email: %s" % (name, email))
```

example: sorting alphabetically and by frequency

```
import nltk

def compute_word_frequencies(filename):
    ...
    return frequencies

freqs = compute_word_frequencies("test.txt")

word_freq_pairs = freqs.items()

for word, freq in sorted(word_freq_pairs):
    print("%s: %s" % (word, freq))

for word, freq in sorted(word_freq_pairs, key=freqs.get,
                         reverse=True):
    print "%s: %s" % (word, freq)
```

overview

recap files, dictionaries, sorting

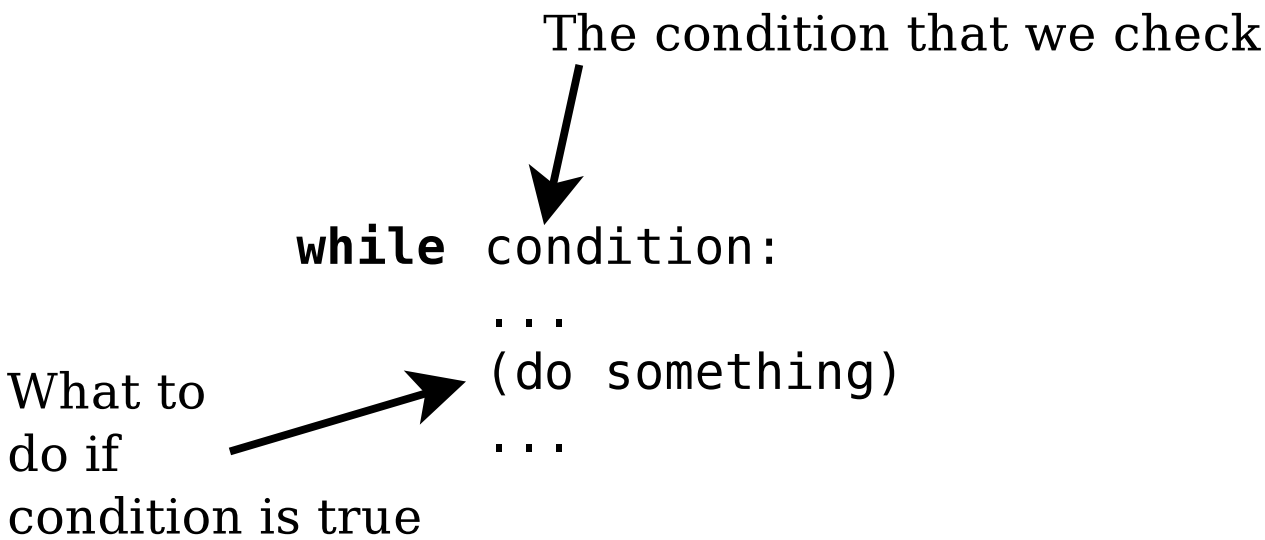
while loops and recursion

higher-order functions

classes and objects

more about looping: `while`

- a `while` loop looks just like an `if`: it executes a block of code if a condition is true
- the difference: `while` will do it again and again until the condition is false
- for instance: loop forever with `while True`



example: reading user input

- the builtin function `input` reads a line from the user

```
line = input()
while line != 'quit':
    print("The line is: %s" % line)
    line = input()
```

break and continue

- break interrupts an ongoing for or while loop
- continue interrupts the current step and goes to the start of the block

```
while True:
    line = input()
    if line == 'quit':
        break
    if line == 'ignore':
        continue
    print("The line is: %s" % line)
```

one more way to repeat: recursion

- **recursion**: a function that calls itself
- why does this work – why doesn't it go on forever?
- a recursive function `f` contains at least two parts:
  - a **base case**: if the input is simple enough, the return value can be computed without further recursion
  - a **recursive call**: the function `f` calls itself with a **simpler** thing as an input
- the typical use of recursion is in **nested** data structures: trees, lists in lists, ...

example: summing a nested list of numbers

- use `isinstance(x, t)` to test if the value `x` is of the type `t`

```
def sum_nested(x):
    if isinstance(x, list):
        sum = 0
        for item in x:
            sum += sum_nested(item)
        return sum
    else:
        return x

testlist = [1, 4, [3, 8], [7, [2, 6], 9], 11]
print(sum_nested(testlist))
```

example: depth of a nested list of numbers

```
def nested_list_depth(x):
    if isinstance(x, list):
        maxdepth = 0
        for item in x:
            d = nested_list_depth(item)
            if d > maxdepth:
                maxdepth = d
        return maxdepth + 1
    else:
        return 0

testlist = [1, 4, [3, 8], [7, [2, 6], 9], 11]
print(nested_list_depth(testlist))
```

example: the factorial function

- the *factorial* function is defined

$$n! = 1 \cdot \ldots \cdot n$$

```
def for_factorial(n):
    product = 1
    for number in range(1, n+1):
        product = product * number
    return product

def rec_factorial(n):
    if n <= 1:
        return 1
    else:
        return n * rec_factorial(n-1)

print(for_factorial(6))
print(rec_factorial(6))
```

- if you can use `for` instead, do it!

summary: different types of looping / repetition

four different ways to do things repeatedly, ordered from simplest to most complex and powerful:

- list comprehension: `[ f(x) for x in some_list ]`
  - transforming a list
- `for`:
  - going through all members in a given collection
  - doing something a fixed number of times: `range(N)`
- `while`:
  - doing something an unspecified number of times (or forever)
- recursion:
  - processing tree-structured or nested data

overview

recap files, dictionaries, sorting

while loops and recursion

higher-order functions

classes and objects

functions with other functions as input

- a function that takes another function as an input is called a **higher-order function**
- example: `sorted(list_of_strings, key=len)`
- example: `do_twice` is a higher-order function, `print_twice` isn't

```
def do_twice(f, x):
    f(x)
    f(x)

def print_twice(something):
    print(something)
    print(something)

do_twice(print, 'hello')

print_twice(len('hello'))
```

example: maximizing w.r.t. some given function

- we have some items in a list and we want to find the maximum according to some measure
- but the measure will be defined by the user!

```
def max_by(collection, measure):
    max_item = None
    max_value = None
    for item in collection:
        value = measure(item)
        if max_value == None or value > max_value:
            max_item = item
            max_value = value
    return max_item

strings = ["this", "is", "a", "list", "of", "strings"]
print(max_by(strings, len))
```

example: processing words

```
import nltk

def print_words(filename, sen_splitter, word_splitter):
    with open(filename) as f:
        content = f.read()
        for sen in sen_splitter(content):
            for word in word_splitter(sen):
                ...

eng_sen_splitter = nltk.tokenize.sent_tokenize
eng_word_splitter = nltk.tokenize.word_tokenize
print_words("english.txt", eng_sen_splitter, eng_word_splitter)

chi_sen_spliter = ...
chi_word_spliter = ...
print_words("chinese.txt", chi_sen_splitter, chi_word_splitter)
```

Chinese word segmentation

- in Chinese, word splitting is not trivial:

example borrowed from Liang Huang

anonymous functions

- sometimes we need “throwaway” functions whose only purpose is to be used with a higher-order function
- we can use **anonymous functions** using the keyword `lambda`

```
lst = ['this', 'is', 'a', 'test', '.']

def number_of_t(s):
    return s.count('t')

print(sorted(lst, reverse=True, key=number_of_t))

print(sorted(lst, reverse=True, key=lambda s: s.count('t')))
```

overview

recap files, dictionaries, sorting

while loops and recursion

higher-order functions

classes and objects

recap from lecture 3: classes and objects

- programmers can define their own types
  - user-defined types are called **classes**
  - the values are called **objects**
- for instance, NLTK defines many classes
- you have already used one such class: `Synset`
- each object contains its own **attributes** and **methods**
  - `x.attr`
  - `x.method(inputs)`

example: representing persons

- assume we have a class `Person` that represents some properties of a person
- every person has a few attributes
  - weight, height, temperature
- the method `get_temperature` returns the temperature of the person
- the method `compute_bmi` computes the body mass index

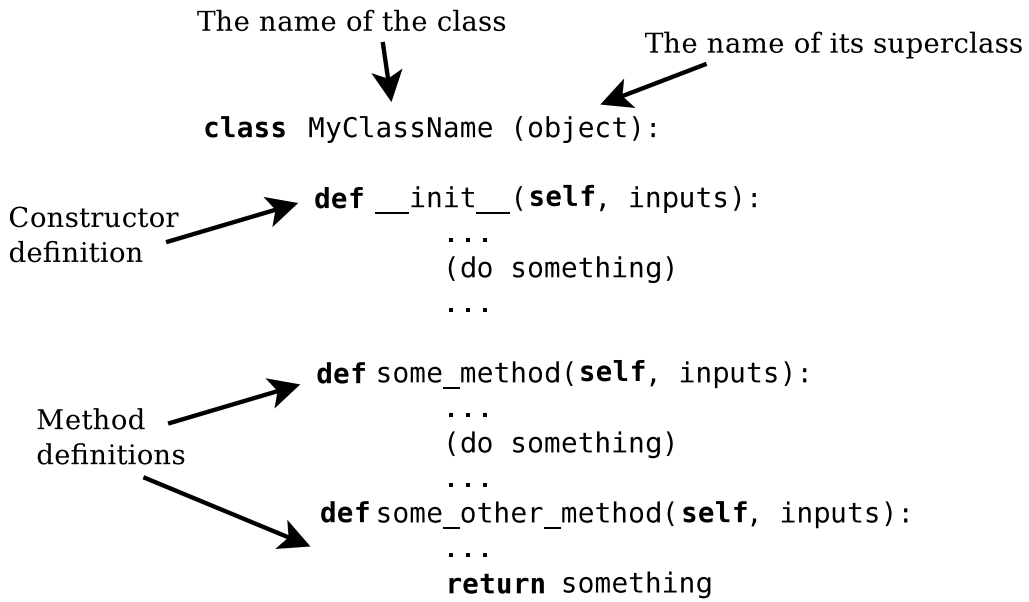
```
joe = Person(weight=80, height=175, temperature=37)

print(joe.temperature)

print(joe.get_temperature())
print(joe.compute_bmi())
```

defining your own classes

- you declare a class using the `class` keyword
- methods are written inside the class and defined with `def`
- note: the first input of each method is called `self` and refers to the current object
- the special method `__init__` is called the **constructor** and is called when an object is created



example: a class describing properties of a person

```
class Person (object):
    def __init__(self, w, h, t):
        self.weight = w
        self.height = h
        self.temp = t

    def get_temperature(self):
        return self.temp

    def compute_bmi(self):
        meters = self.height / 100
        bmi = self.weight/(meters*meters)
        return bmi

john = Person(80, 175, 37)
print(john.compute_bmi())
```

example: address book

- assume we have a class `AddressBook` that contains the method `lookup`
- `lookup` returns an object of the type `PersonData`
- `PersonData` contains the attributes `name`, `email`, `phone`, `birthday`, ...

```
addressbook = ...

richards_data = addressbook.lookup("Richard")

print(richards_data.birthday)
```

example: the person database

- the class `PersonData` is an example of a class that just holds some data: no methods except the constructor
- typical use of the constructor: setting initial values of the attributes

```
class PersonData(object):
    def __init__(self, n, e, p, b):
        self.name = n
        self.email = e
        self.phone = p
        self.birthday = b

addressbook = ...
richards_data = addressbook.lookup("Richard")
print(richards_data.birthday)
```

example: address book

- we create new objects of a class using the class name, e.g. `PersonData(...)` and `AddressBook()`

```
class AddressBook(object):
    def __init__(self):
        self.database = {}
        ...
        self.database["Richard"] = PersonData("Richard",
                                              "some_email@gu.se",
                                              "031-7864418",
                                              "July 9")
    def lookup(self, name):
        return self.database[name]

addressbook = AddressBook()
richards_data = addressbook.lookup("Richard")
print(richards_data.birthday)
```

why classes and objects?

- we could have implemented the address book using a dictionary instead of `AddressBook` and a tuple instead of `PersonData`
- . . . but our solution is more understandable because the class definitions tell what we mean
- just like we divide the **code** into separate functions to make it manageable, we divide our **data** into separate objects
- more about object-oriented design in the next lecture

programming paradigms

- **object-oriented** programming styles and programming languages emphasize classes and objects
  - they build abstractions around the data
  - examples of object-oriented languages: Java, C++
- **functional** styles and languages emphasize higher-order functions, lambdas, and recursion
  - they build abstractions by combining functions
  - examples of functional languages: Haskell, ML, Lisp
- Python is a pragmatic object-oriented language but includes some features from functional languages: higher-order functions and lambdas

next two lectures

- lecture 6: more object-oriented programming; a little bit about the theory of algorithms and data structures
- lecture 7: mainly course recap, example exam