1_Introduction

February 21, 2024

Programming Techniques in Computational Linguistics II – FS24

1 Lecture 1: Introduction

1.1 Topics

- Admin:
  - Teaching team
  - Course overview
  - Exercises & Exams
  - Tools used in this class
- Coding refresher:
  - Syntax
  - Data types
  - Namespaces
  - Coding guidelines

1.2 Learning Objectives

You know …

- what to expect from this class
- what is required of you to successfully complete this course
- where to find information and additional help
- basic concepts of coding with python

2 Admin

2.1 Lecturers

Andreas Säuberli

Lukas Fischer

Contact via OLAT: use the *@Lecturers* button.

2.2 Tutors

- Isabelle Cretton
- Lucas Suomela

- Rong Li
- Tosca Peruzzi-Vieli

Contact via OLAT: use the *@Tutors* button.

2.3 OLAT

- Schedule
- Slides & Code
- Forum
- Feedback from Assignments and Exams

2.4 Grading

- 25% Exercise grade
- 25% Midterm exam
- 50% Final exam
- Detailed information on OLAT

3 Exercises

- Weekly exercises to practice what you learned in the lectures
- Three exercises are graded, the rest are optional
- Working in groups is not permitted (graded exercises)
- Exercises are distributed and submitted via GitLab
- Grading:
  - Exercise 2 and exercise 6 are worth 1 point each
  - Exercise 8 is worth 3 points
  - `exercise_grade = sum(exercise_points) + 1`
- Help: Tutorial, OLAT forum, email the tutors

3.1 Midterm Exam

- When: 24.4.2024, 10:15-11:00
- Where: AND-3-02/06 (this room)
- **In-person attendance is mandatory**
- Tools: pen and paper, no additional materials allowed

3.2 Final Exam

- **Take-home exam**
- Duration of 24h (12.6.2024,10:00 until 10:00 the next day)
- Multiple programming assignments
- Download and submission via OLAT
- Tools: slides, documentation, online sources
- No team work, no ghostwriting
- In cases of doubt: oral repetition exam

3.3 Lecture Format

3.3.1 Lecture Sessions:

- Theory
- Practical exercises
- Interactive examples

3.3.2 Format:

- Sessions will be recorded (but no guarantees!)
- On site: AND-3-02/06

3.3.3 Tutorial Sessions

- **The first tutorial is this Friday!**
- The tutors will show you how to use Git/GitLab in your assignments.
- It is highly recommended to attend!
- On site: AND-3-06

3.3.4 Lecture Slides

- Make use of Jupyter Notebook and RISE
- Will be uploaded as jupyter notebooks and as PDF scripts/slides
- To run notebook in jupyter:

```
conda install ipython jupyter # Install with conda
pip install --upgrade ipython jupyter # Install with pip
```

- Install RISE (optional):

```
conda install -c conda-forge rise # Install with conda
pip install RISE # Install with pip
```

- Navigate to directory containing the notebook (`filename.ipynb`)
- Start running a notebook server:

```
jupyter notebook
```

- Open/create notebooks in browser
- Or: Open in Editor (e.g. Visual Studio Code)
- Basic tutorial

3.3.5 GitLab

- Access and submit solutions of assignments on GitLab
- Create an account with your UZH email address
- Visit the tutorial on Friday for an introduction on how to use GitLab!

3.3.6 Git

- version control system
- collaboration
- open source

3.3.7 Git Demonstration

- `git clone https://gitlab.uzh.ch/lukas.fischer3/git_demo`
- `git pull` #make sure your local code is up to date
- #Edit or add files
- `git add <filename>` #add file to the staging area
- `git status` or `git diff` #review changes
- `git commit -a -m "meaningful commit message"` #commit files to local git repository
- `git push` #push changes to remote repository

3.3.8 Coding with AI-tools

- You may use AI-tools like ChatGPT in the exercises and final exam.
- However, in order to learn from the exercises, we highly recommend **solving them by yourself first**.
- You should use AI-tools to help you understand code or fix errors. Example prompts:
  - How does this code work? Explain step by step.
  - I wrote this function for XYZ, but the output is A instead of B. Why?
  - I got this cryptic error message: …

4 Coding Refresher

4.1 Python

Make sure to use version `>=3.9`!

4.1.1 Script language

- High-level
- Interpreted
- Interactive

[ ]:
```
string = "This is an interactive python environment"
```

[ ]:
```
print(string + "!" * 5)
```

4.1.2 Pardigms

- object-oriented
- imperative
- functional

4.1.3 Typing

- Dynamic
- Duck-typing (“If it walks like a duck, and it quacks like a duck, then it must be a duck.”)

[ ]:
```
# java
int i = 10;
```

Duck-typing: “+”-Operator (`__add__`)

[ ]:
```
5 + 6
```

[ ]:
```
"Hello" + " world"
```

[ ]:
```
class Hobbit():
    def __init__(self, name):
        self.name = name
    def __add__(self, b):
        return f'{self.name} and {b.name}'
```

[ ]:
```
Hobbit("Frodo") + Hobbit("Sam")
```

Duck-typing: `len()`

[ ]:
```
len("Hello")
```

[ ]:
```
len([1, 2, 3])
```

[ ]:
```
len(10)
```

[ ]:
```
len(Hobbit("Frodo"))
```

4.2 Syntax Elements

4.2.1 Keywords:

- reserved, cannot be used as variable names, etc.
- examples: `class def for in is not with as True False`

4.2.2 Symbols

- human readable
- examples: `. , : @ ( ) { } [ ] < = > + - * / | \ &`

4.2.3 Identifiers

- names of variables, classes, functions, libraries
- examples: `text`, `Spammer`, `get_tokenized`, `_ignored`, `HTML5Parser`

- **Rules:**
  - contain letters, numbers and underscores
  - never start with a number

- cannot be a keyword
- is case-sensitive (upper / lower case matters)

4.2.4 Literals

- representing a fixed value
- often stored in variables
- examples on the following slides

Numbers:

```
[ ]: # integer
     10
```

```
[ ]: # float
     10.
```

```
[ ]: # exp. notation
     1.23e-4
```

```
[ ]: # hex notation:
     0x4E20
```

Strings In Python 3, Strings are Unicode by default.

```
[ ]: 'string with single quotes'
```

```
[ ]: "double quotes  "
```

```
[ ]: """multi
     line"""
```

```
[ ]: # raw strings:
     r'[^\W\d]\w*'
```

```
[ ]: # Python 2 compatibility:
     u'alte Zöpfe'
```

```
[ ]: # f-String (Python 3.6+):
     count = 17
     location = "Downloads"
     print(f'found {count} items in {location}')
```

```
[ ]: items = [2,4,8,3,2,2,3]
     f'avg.: {sum(items)/len(items):.2f}'
```

4.2.5 Operators

- Keywords: `not`, `and`, `or`, `in`, `is`
- Symbols: `+`, `-`, `<`, `==`, `!=` …

- Augmented Assignment: `+=`, `-=`, `**=`

[ ]:
```
3 + 3.0
```

[ ]:
```
'spam ' + 'spam'
```

[ ]:
```
['spam', 'spam'] + ['spam']
```

[ ]:
```
3 * 3.0
```

[ ]:
```
'spam ' * 5
```

[ ]:
```
['spam'] * 3
```

Precedence

[ ]:
```
def is_duck(walks: bool, swims: bool, quacks: bool) -> bool:
    return bool(walks and (swims or quacks))
```

[ ]:
```
is_duck(walks=False, swims=False, quacks=True)
```

- **Rules of precedence**
  - Arithmetic expressions: **P**lease **E**xcuse **M**y **D**ear **A**unt **S**ally (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction)
  - Comparison operators are weaker than arithmetic operators
  - `not`, `and` and `or` are the weakest
- In cases of doubt: Use parentheses!

4.2.6 Klicker Quiz

https://pwa.klicker.uzh.ch/join/lfische

```
def contains_string(x: str) -> bool:
    '''Return true if input string is contained in y or z'''

    y = "supercalifragilisticexpialidocious"
    z = "baba"

    return x in y or z

contains_string("b")
```

Example: What is the output?

[ ]:
```
not False or False and True
```

In this example, Python will only evaluate `not False`, since the right condition after the `or` is irrelevant.

This can save you time: put a statement that takes a long time to evaluate to the right of `or`. If the left condition is true, the time consuming condition doesn't have to be evaluated!

example: `if (i < 100) or ("hello" in very_large_list)`

[ ]:
```python
# demonstration
import time
bool(time.sleep(3))
```

[ ]:
```python
for _ in range(10):
    print(time.sleep(1) or True)
```

[ ]:
```python
for _ in range(10):
    print(True or time.sleep(1))
```

Example: What is the output?

[ ]:
```python
def contains_string(x: str) -> bool:
    '''Return true if input string is contained in y or z'''

    y = "supercalifragilisticexpialidocious"
    z = "baba"

    return x in y or z
```

[ ]:
```python
contains_string("b")
```

Ask ChatGPT: “The function returns”baba”, why?”

> The function `contains_string` is designed to check if the input string `x` is contained within either the string `y` or the string `z`. However, the implementation has a **logical flaw**. Currently, the return statement is written as `return x in y or z`. Here, `z` is not being compared to `x`. Instead, Python treats `z` as a boolean value, and since it’s a non-empty string, it **evaluates to `True` in boolean context**. If `x` is not in `y`, it directly returns the value of `z`, which is the string `"baba"`. This is not the desired behavior. To fix this issue and return `True` if `x` is in either `y` or `z`, you need to ensure that the return statement checks both conditions separately.

4.3 Data Types

4.3.1 Basic Data Types

- **Booleans**
- **Numbers**: Integers, Floats
- **Sequence Containers**: Strings, Lists, Tuples, Byte Strings, Byte Arrays
- **Set Containers**: Sets, Frozensets
- **Mapping Container**: Dictionaries

4.3.2 Mutable vs Immutable Types

Many basic data types are immutable; they cannot be changed after creation. Immutable types include:

```
- booleans, integers, floats, strings, tuples, byte strings, frozensets
```

Mutable types include:

```
- lists, byte arrays, sets, dictionaries
```

[ ]:
```
string_seq = "hello world"
string_seq[0] = "H"
```

[ ]:
```
list_seq = [1, 2, 3, 4]
list_seq[0] = 55
list_seq
```

4.3.3 Built-in Functions

- Essential and useful functions
  - `open()`
  - `print()`
  - `len()`
  - `min()` / `max()`
  - `range()`
- Creation/conversion of built-in types:
  - `str()`
  - `int()`, `float()`
  - `list()`, `set()`, `dict()`

4.3.4 Standard library

Extensive collection of modules for concrete application scenarios that are provided, e. g.

```
- collections
- itertools
- math
- random
- datetime
- zipfile
- pathlib
- urllib
```

Default dictionary

[ ]:
```
from collections import defaultdict
# default values for new dictionary keys

d = defaultdict(list) # default value is []

for index, character in enumerate("programming"):
    d[character].append(index)

d
```

Counter

[ ]:

```
from collections import Counter
# subclass of dictionary

c = Counter("computational linguistics")
c.total()
```

[ ]:

```
c.most_common()
```

Datetime

[ ]:

```
from datetime import datetime
# date-time representation

exam_date = datetime(2024, 6, 12, 10, 0)
time_left = exam_date - datetime.now()
time_left.days
```

4.3.5 Interactive Part: Slicing

Klicker Quiz 2 Sequences : Accessing elements of a container

https://pwa.klicker.uzh.ch/join/lfische

[1]:

```
def foo(sequence):
    print(sequence[0])
    print(sequence[-1])
    print(sequence[0:1])
    print(sequence[::2])
    print(sequence[::-1])
```

[2]:

```
foo([1, 2, 3])
```

```
1
3
[1]
[1, 3]
[3, 2, 1]
```

[ ]:

```
foo('123')
```

[ ]:

```
foo("Frodo")
```

4.4 Namespaces

Module *spam.py*

```
import random

SPAM_AMOUNT = 0.3

def spam(text, amnt=SPAM_AMOUNT):
    'Insert given amount of spam.'
    spammer = Spammer('SPAM!')
    return spammer.spam(text, amnt)

class Spammer:
    ...
    def spam(self, text, amnt):
        "Insert spam randomly."
        ...
```

[ ]:
```
import spam
```

[ ]:
```
spam.spam
```

[ ]:
```
spam.Spammer.spam
```

4.4.1 Import Statements

[ ]:
```
import spam
spammer = spam.Spammer('SPAM!')
print(spammer.spam("This is an example", 1))
print(spam.spam("This is another example"))
```

[ ]:
```
from spam import Spammer
Spammer('SPAM!').spam("This is an example", 0.5)
```

Advantages and disadvantages of the two approaches?

Other possibilities:

[ ]:
```
from spam import Spammer as spm, SPAM_AMOUNT as amt
from spam import * # namespace cluttering!
```

But not:

[ ]:
```
import spam.Spammer
```

4.5 Coding Guidelines

4.5.1 Importability

	Script to run	Module to import
	`$ python3 example.py`	import example
Purpose	Applying the functionality	Providing the functionality
Behaviour	All instructions on the top level are executed.	All instructions on the top level are executed.
End	Main call on top level.	On top level only definitions,no instructions.

example.py:

```python
def run(infile, outfile):
    for line in infile:
        ...

run(sys.stdin, sys.stdout) # we don't want to run this if imported as module
```

4.5.2 Script and module in one

Convention: - Main call to script in function `main()` without arguments - Script or not? Checking a specific variable at the end of the module - `main()` is only executed with positive script check - No side effects when only importing as module (only loading, no processing)

```python
def main():
    run(sys.stdin, sys.stdout)

def run(infile, outfile):
    for line in infile:
        ...

if __name__ == '__main__':
    main()
```

4.5.3 Code Documentation

Important: **Code is read more often than written.**

Good Python code is readable on its own. However, it is essential to add targeted information in natural language:

- Short summary of functionality of the individual components
- Indications for using the code (arguments, effects)
- Explanation of complex code parts

Language: maximum portability if English.

4.5.4 Type Hints

```python
def spam(text: str, amnt: float = 0.3) -> str:
    ...
```

- Annotation of types of arguments and return values
- Ignored by the interpreter
- Intended for documentation purposes
- … and / or static type checks with external tools

Nesting

```python
def most_common(words: list[str], n: int = 5) -> list[tuple[str, int]]:
    '''Get the top-n words and their frequency.'''
```

Many common types are defined in the `typing` module:

```
from typing import Iterable, Sequence, TextIO, Any, Optional, ...
```

Some degree of freedom: gradual typing, sub-specifications, customtypes:

```
Any, Optional[str], MySpammer, "ClassDefinedLater"
```

4.5.5 Docstrings

Every module, function and class should have a docstring. Docstrings are intended for people *using* the code.

A docstring contains:

- A short sentence, explaining the purpose of the module / function …
- Possibly other details that are useful for the user, e.g.:
  - Non-obvious side effects
  - Meaning of the individual arguments
  - Details about return values

Docstring example:

```
def word_tokenise(sent: str) -> List[str]:
    '''
    Split a sentence into word tokens.
    Args:
        sent: a sentence including
            sentence-final punctuation
    Returns:
        a flat list of tokens
    '''
    ...
```

4.5.6 Comments

Comments explain certain parts in the code in more detail. They are intended for people adapting the code.

Comments explain:

- Non-obvious details
- Complex instructions
- Uncertainties
- Notes for later (TODO, FIXME)
- Tricks and hacks (better to avoid those alltogether)
- Workarounds
- …

4.5.7 Bad Coding Example

[ ]:

```
from random import *

def do_something(arg1):
    i = randint(1, 10)
    return arg1 * i

str = do_something('hello')
```

4.6 Take-home messages

- OLAT: general information, slides, forum, contact lecturers/tutors, feedback/grades
- GitLab: for exercises. Create an account, complete Exercise 0, and visit the tutorial.
- Coding refresher: You know the basics of programming with python, including best practices for writing code.