



Python (2/3)

PU Deep Learning

Nils Reiter, nils.reiter@uni-koeln.de

April 27, 2021 (Summer term 2021)

Recap

- ▶ Git
 - ▶ Branches, Remotes
- ▶ Python
 - ▶ Running Python scripts
 - ▶ Python syntax
 - ▶ Data types
 - ▶ Standard library
- ▶ Exercise 2
 - ▶ Registrierung der Studienleistung → Ilias
 - ▶ GitHub

Ergebnisse der IDH-Umfrage

Ergebnisse der IDH-Umfrage

Beratungsangebote sind da, damit man sie wahrnimmt

- ▶ Alle Dozent:innen bieten Sprechstunden an: Termin nach Absprache und online
- ▶ Studienberatung
 - ▶ Fachspezifisch: Jürgen Hermes, IDH
<https://dh.phil-fak.uni-koeln.de/mitarbeiterinnen/wissenschaftliche-mitarbeiterinnen/dr-juergen-hermes>
 - ▶ Fachübergreifend: <https://verwaltung.uni-koeln.de/abteilung21/>
- ▶ Schreibzentrum der Philosophischen Fakultät <https://schreibzentrum.phil-fak.uni-koeln.de>
- ▶ Psychologische Beratung: **Studierendenwerk**
<https://www.kstw.de/beratung/psychologische-beratung>
- ▶ Gewaltschutz
 - ▶ <http://www.hilfetelefon.de>
 - ▶ Notunterkünfte im Studierendenwerk: <https://www.kstw.de>
 - ▶ Sozialdienst katholischer (Frauen|Männer) <https://www.skf-koeln.de>|<https://www.skm-koeln.de>


Today

List Comprehension

~~Classes and Functions~~

Input/Output

Exercise

 Slides will only scratch the surface, reading documentation is necessary!

Section 1

List Comprehension

Background: Defining Sets

Enumeration

- ▶ Explicit listing of set elements: $S = \{1, 5, 7, 9\}$

Background: Defining Sets

Enumeration

- ▶ Explicit listing of set elements: $S = \{1, 5, 7, 9\}$
- ▶ Sometimes, with ellipsis: $S = \{1, 2, 3, \dots, 100\}$
 - ▶ Assumption: Recognizable pattern

Background: Defining Sets

Enumeration

- ▶ Explicit listing of set elements: $S = \{1, 5, 7, 9\}$
- ▶ Sometimes, with ellipsis: $S = \{1, 2, 3, \dots, 100\}$
 - ▶ Assumption: Recognizable pattern

Background: Set-builder notation

$$S = \{x \mid p(x)\}$$

- ▶ $|$: »such that«; $p(x)$: predicate

Background: Defining Sets

Enumeration

- ▶ Explicit listing of set elements: $S = \{1, 5, 7, 9\}$
- ▶ Sometimes, with ellipsis: $S = \{1, 2, 3, \dots, 100\}$
 - ▶ Assumption: Recognizable pattern

Background: Set-builder notation

$$S = \{x \mid p(x)\}$$

- ▶ \mid : »such that«; $p(x)$: predicate
- ▶ S contains all values of x for which $p(x)$ returns true

Background: More Complex Predicates

- ▶ Logical and: $S = \{x \mid \underline{p_1(x)} \wedge \underline{p_2(x)}\}$
 - ▶ S contains all values of x for which $p_1(x)$ and $p_2(x)$ is true

Background: More Complex Predicates

- ▶ Logical and: $S = \{x \mid p_1(x) \wedge p_2(x)\}$ *und*
 - ▶ S contains all values of x for which $p_1(x)$ and $p_2(x)$ is true
- ▶ Logical or: $S = \{x \mid p_1(x) \vee p_2(x)\}$ *oder*
 - ▶ S contains all values of x for which $p_1(x)$ or $p_2(x)$ is true

Background: More Complex Predicates

- ▶ Logical and: $S = \{x \mid p_1(x) \wedge p_2(x)\}$
 - ▶ S contains all values of x for which $p_1(x)$ and $p_2(x)$ is true
- ▶ Logical or: $S = \{x \mid p_1(x) \vee p_2(x)\}$
 - ▶ S contains all values of x for which $p_1(x)$ or $p_2(x)$ is true
- ▶ Domain of x : $S = \{x \mid x \in X \wedge p(x)\}$
 - ▶ S contains all elements of X , for which $p(x)$ is true

Background: More Complex Predicates

- ▶ Logical and: $S = \{x \mid p_1(x) \wedge p_2(x)\}$
 - ▶ S contains all values of x for which $p_1(x)$ and $p_2(x)$ is true
- ▶ Logical or: $S = \{x \mid p_1(x) \vee p_2(x)\}$
 - ▶ S contains all values of x for which $p_1(x)$ or $p_2(x)$ is true
- ▶ Domain of x : $S = \{x \mid x \in X \wedge p(x)\}$
 - ▶ S contains all elements of X , for which $p(x)$ is true

Examples

$$\begin{aligned}
 \{x \mid x + x = x * x \wedge x \in \mathbb{N}\} &= \{2\} \\
 \{x \mid x \in \mathbb{N} \wedge x < 5\} &= \{1, 2, 3, 4, 5\} \\
 \{x \mid x = y * 2 \wedge y \in Y\} &= \{2, 4, 6\} \quad \text{if } Y = \{1, 2, 3\}
 \end{aligned}$$

List Comprehension

- ▶ Python-specific!
- ▶ Special syntax for specifying lists
- ▶ Similar to set-builder notation

List Comprehension

- ▶ Python-specific!
- ▶ Special syntax for specifying lists
- ▶ Similar to set-builder notation

Lists in Python

```
1 # an empty list
2 x = []
3 # a list with two elements
4 x = [1, 2]
5 # a list with three elements of different types
6 x = [1, True, "Hello"]
```

enumeration

List Comprehension

- ▶ Python-specific!
- ▶ Special syntax for specifying lists
- ▶ Similar to set-builder notation

Lists in Python

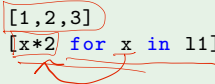
```
1 # an empty list
2 x = []
3 # a list with two elements
4 x = [1, 2]
5 # a list with three elements of different types
6 x = [1, True, "Hello"]
```

Enumeration of list elements

List Comprehension

Examples

```
1 l1 = [1,2,3]
2 l2 = [x*2 for x in l1] # [2,4,6]
```




$L2 = []$
for x in $L1$:
 $L2.append(x \cdot 2)$
 $\#L2 = [2, 4, 6]$

List Comprehension

Examples

```
1 l1 = [1,2,3]
2 l2 = [x*2 for x in l1] # [2,4,6]
3
4 l1 = ["the", "dog", "barks"]
5 l2 = [x for x in l1 if "e" in x] # ["the"]
```



List Comprehension

Formal

`[expr for x in list for/if ...]`

optional

- ▶ Expression `expr` is evaluated in context of for loop
 - ▶ Expression can be full-fledged python expression
e.g., another list comprehension!
- ▶ One for loop is required, more are optional
- ▶ `expr` often uses `x`, but doesn't have to
- ▶ `list` needs to be a list (or coercible into one)
- ▶ Evaluates to a list (and the whole thing is an expression)

demo

Section 2

Classes and Functions

Functions

- ▶ Sub-programs in your program
- ▶ Built-in functions: `print(...)`, `input(...)`, `str(...)`, ...
- ▶ Custom functions: Defined by yourself

Functions

- ▶ Sub-programs in your program
- ▶ Built-in functions: `print(...)`, `input(...)`, `str(...)`, ...
- ▶ Custom functions: Defined by yourself

```
1 def myFunction(arguments):  
2     # execute code ←  
3     return
```

my function (args)

Function Arguments

- ▶ Arguments are (like everything else) dynamically typed
- ▶ Multiple arguments are listed with a comma

Function Arguments

- ▶ Arguments are (like everything else) dynamically typed
- ▶ Multiple arguments are listed with a comma

```
1 def myFunction(arg1, arg2, arg3):  
2     # do something  
3     return
```

Function Arguments

Type checking and default values

Listing: Type Checking

```
1 def myFunction(arg1, arg2):
2     if not isinstance(arg1, str):
3         # throw exception
4         raise TypeError
5         # do something ←
6     return ←
```

```
def add(x, y):
    return x + y
```

```
add(True, "Hallo")
```

Function Arguments

Type checking and default values

Listing: Type Checking

```
1 def myFunction(arg1, arg2):  
2     if not isinstance(arg1, str):  
3         # throw exception  
4         raise TypeError  
5     # do something  
6     return
```

```
1 def myFunction(arg1, arg2="bla"):  
2     print(arg2)  
3     return
```

myFunction("Hallo", "blubb")

Function Arguments

Named arguments

Argument named can be used when calling a function

```
1 def myFunc(a1, a2):  
2     # do something  
3     return  
4  
5 myFunc(1,2) # a1 = 1, a2 = 2  
6 myFunc(a2=1, a1=2) # a2 = 1, a1 = 2
```

Variable Arguments

Argument values as list

We can define functions without a fixed number of arguments:

```
1 def myFunc(*args):
2     # Inside the function, args is a list
3     numArgs = len(args)
4     print(str(numArgs) + " arguments were given.")
5     print("the first argument is " + str(args[0]))
6     return
7
8 myFunc(1,2,3)
9 myFunc(1,2)
```

Variable Arguments

Named arguments as dictionary

```
1 def func(**kwargs):
2     # inside the function, kwargs is a dictionary
3     for key in kwargs:
4         print(key + ": " + str(kwargs[key]))
5     return
6
7 func(a=1, b="Hallo", c=7)
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
4
5 # return value is None
6 def f2():
7     return
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
4
5 # return value is None
6 def f2():
7     return
8
9 # return value is None
10 def f3():
11     return None
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
4
5 # return value is None
6 def f2():
7     return
8
9 # return value is None
10 def f3():
11     return None
12
13 # return value is None
14 def f4():
15     x = 3
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
4
5 # return value is None
6 def f2():
7     return
8
9 # return value is None
10 def f3():
11     return None
12
13 # return value is None
14 def f4():
15     x = 3
```

```
1 # return value is a list
2 def f5():
3     return [5]
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
4
5 # return value is None
6 def f2():
7     return
8
9 # return value is None
10 def f3():
11     return None
12
13 # return value is None
14 def f4():
15     x = 3
```

```
1 # return value is a list
2 def f5():
3     return [5]
4
5 # return value is a dictionary
6 def f6():
7     return {"value": 5}
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
4
5 # return value is None
6 def f2():
7     return
8
9 # return value is None
10 def f3():
11     return None
12
13 # return value is None
14 def f4():
15     x = 3
```

```
1 # return value is a list
2 def f5():
3     return [5]
4
5 # return value is a dictionary
6 def f6():
7     return {"value": 5}
8
9 # return value is a list
10 def f7():
11     return [None]
```

Terminating a Function

- ▶ We can terminate the function with the `return` statement

```
1 # return value is 5
2 def f1():
3     return 5
4
5 # return value is None
6 def f2():
7     return
8
9 # return value is None
10 def f3():
11     return None
12
13 # return value is None
14 def f4():
15     x = 3
```

```
1 # return value is a list
2 def f5():
3     return [5]
4
5 # return value is a dictionary
6 def f6():
7     return {"value": 5}
8
9 # return value is a list
10 def f7():
11     return [None]
12
13 # an empty function body also returns None
14 def f8():
15     pass
```


None

- ▶ `None` is the null-value in Python
- ▶ The type of `None` is `NoneType`
- ▶ `NoneType` only has a single possible value: `None`

None

- ▶ `None` is the null-value in Python
- ▶ The type of `None` is `NoneType`
- ▶ `NoneType` only has a single possible value: `None`
- ▶ If evaluated, `None` is considered to be false

```
1 if None:  
2     # Never executed  
3     print("Nothing is false")
```

Section 3

Input/Output

Introduction

- ▶ IO: Input/Output
- ▶ Standard model: Streams of bytes/characters: Most programming languages

Introduction

- ▶ IO: Input/Output
- ▶ Standard model: Streams of bytes/characters: Most programming languages
- ▶ Regular workflow
 1. Open a file
 - ▶ Configure: Mode, encoding, file type, ...
 2. Read from file or write to file
 3. Close file

IO in Python

Listing: File Reading

```
1 # open file and create a file
2 # object
3 fh = open(FILENAME)
4
5 # call a method on file object
6 for line in fh.readlines():
7     # process each line
8
9 # close object
10 # (important, because open files
11 # consume resources)
12 fh.close()
```

Listing: File Writing

```
1 # open file and create a file
2 # object
3 fh = open(FILENAME, mode="w")
4
5 # write list content into file
6 for line in listOfResults:
7     fh.write(line)
8
9 # close object
10 # (important, because open files
11 # consume resources)
12 fh.close()
```

IO in Python

`open()`

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

- ▶ <https://docs.python.org/3/library/functions.html#open>
- ▶ mode: Reading or writing
- ▶ buffering: Controls when and in which chunks data is actually written to the disk
- ▶ encoding: Specify encoding (hint: Make sure that UTF-8 is used)
- ▶ errors: How to handle encoding errors
- ▶ newline: Which characters are used to indicate a new line?
- ▶ ...

IO in Python

Methods of the File Object

- ▶ <https://docs.python.org/3/library/io.html#module-io>
- ▶ `close()`: Flush and close the stream
 - ▶ In Java, `close()` only closes without flushing
- ▶ `flush()`: Flush the stream
- ▶ `readline(size=-1)`: Read and return one line from the stream
- ▶ `readlines(hint=-1)`: Read and return a list of lines from the stream.
- ▶ `write(s)`: Write the string `s` to the stream and return the number of characters written.
- ▶ ...

Section 4

Exercise

Exercise 03

`https://github.com/idh-cologne-deep-learning/exercise-03`