Symbolische Programmiersprache - Lecture 1

Prof. Dr. Barbara Plank

Symbolische Programmiersprache
MaiNLP, CIS LMU Munich

WS2025/2026

Willkommen - Welcome!

This course is about (Python) coding, natural language processing (NLP) and some machine learning (ML)

Why Python?

Example: Programming languages on Github

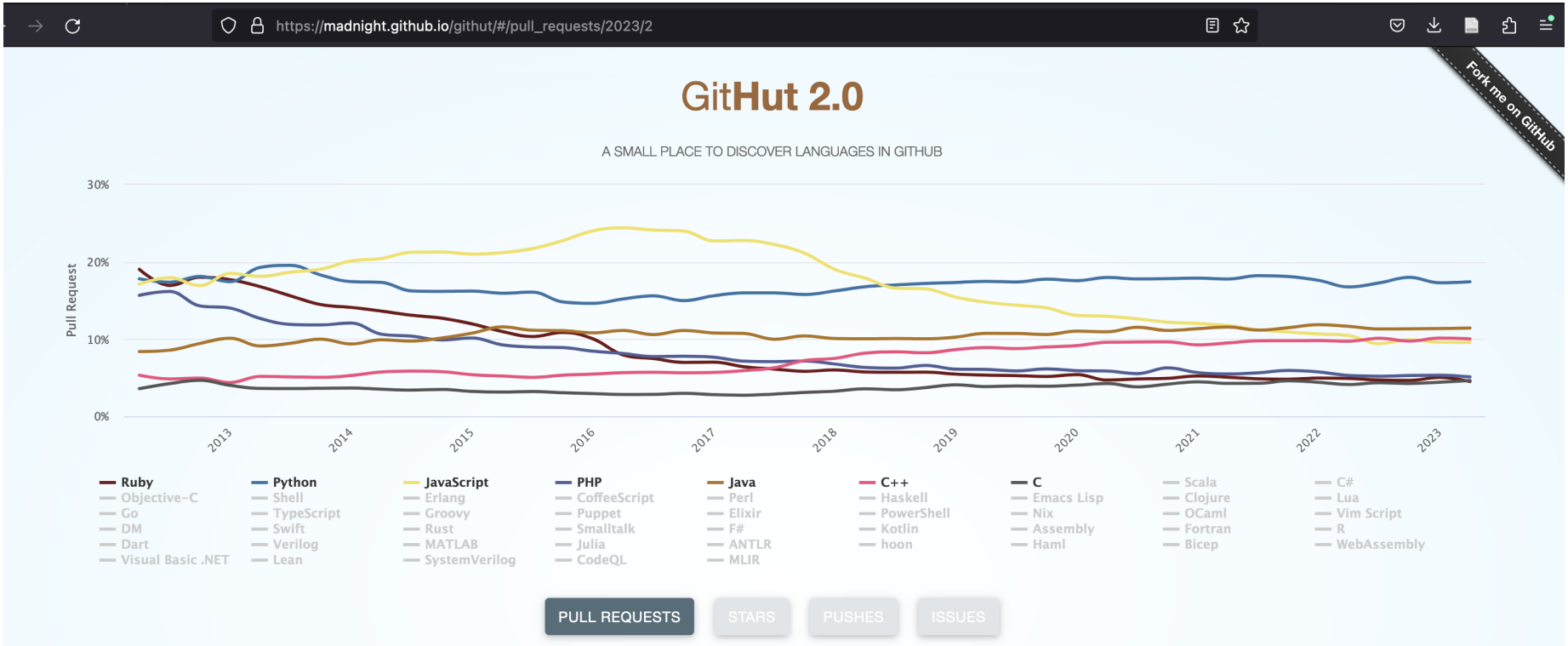


Figure: `https://madnight.github.io/githut/#/pull_requests/2023/2`

Who we are - Lecturers


- Prof. Dr. Barbara Plank and Verena Blaschke

Who we are - Exercises


- Exercises: Dr. Robert Litschko and MEng Beiduo Chen
- Teaching assistant:
  - Katharina Halser

Who are you?

- Go to: menti.com 

Course organization

- Register in LSF and Moodle: “Symbolische Programmiersprache Wintersemester" (Einschreibeschlüssel: SymProg2526)
- Tue **lecture** 16:00 – 18:00 (c.t) $\Rightarrow$ 16.15 - 17.45
- Thu **exercise** 16:00 – 18:00 (c.t) $\Rightarrow$ 16.15 - 17.45
- You are expected to come to class. We stream the lectures over Zoom as secondary channel, e.g. in case you do not feel well do not come to class but attend on Zoom (Note: no recordings)
- ECTS: 6 (3 for lectures, 3 for exercises)
- You can always ask questions in both English and German

Course organization

- How to reach us: We have a **dedicated Slack** workspace.
- Sign up to Slack (the link can also be found on Moodle):

Course organization

Please post questions on Slack (avoid direct emails).

- We give low priority to questions already answered in previous lectures, exercises and posts;
- You are expected to conduct online search for answers before reaching out to us;
- You are highly encouraged to participate and help each other on the channel and during the labs.

The teaching team will check the discussion on slack regularly within normal working hours.

- Do not expect answers late in the evenings and on weekends;
- Start working on your homeworks early;
- Come to the lab sessions and ask questions there.

Course organization

Check Moodle for

- Announcements
- All material for lectures and exercises
- Information on exams

Exam

- Exam (Klausur) - Important: **Thursday 12.02.2026 after 16:00, Oettingenstr.**
- Trial exam in January (see schedule on Moodle)
- Pen and paper exam, no tools allowed
- Two parts, depending which part of the course you take (lecture, exercise or lecture+exercise)
- CS (Informatik) students can only take the lecture part of this course (and exam)

Homeworks (on Gitlab)

- To deepen the materials and prepare for the exam we have **optional weekly homeworks**
  - Homeworks are designed to provide you an opportunity to engage with the course material, and prepare you for the exam.
  - We suggest you to form groups of 3 to 4 people
  - Group creation until next week Thursday at 12.00 (lunch/noon) - before the exercise session - See Moodle for details
  - Suggested deadline to submit homeworks (on gitlab): **Wednesday at 12:00 (lunch/noon)** (day before the next exercise session on Thursday)
  - **Note:** your homework will not be graded (and no bonus points).
  - **Feedback during exercise hours:** During the exercise sessions, we will discuss example solutions and selected homework submissions. Use the weekly moodle feedback forms if your group would like to submit the exercise for potential feedback during the exercises:


Request feedback - homework 1

What does this course cover?

Course content

- **Fundamentals**: Python fundamentals, including (Unit) Tests; Object-oriented programming in Python
- **NLP (and IR) foundations**:
  - How to represent text, corpora and NLP pre-processing in NLTK
  - Syntactic processing
  - Lexical semantic processing
  - Document-level processing (Information Retrieval, IR): web crawling, tf-idf, search engine
- **ML**: Machine Learning, supervised and unsupervised learning algorithms
  - Classification with kNN and Naïve Bayes
  - Clustering with k-means und Brown algorithm

Plan for Today: Python Fundamentals and Testing

- Today's agenda:
  - Python fundamentals
  - Testing in Python (DocTests, UnitTests)
- Lab on Thursday:
  - Intro to git, Pycharm
  - Release homework 1

Python fundamentals and Python Tests

Overview

- Part 1: Python fundamentals
- Part 2: Unit tests

Python

Python is an *interpreted* language (in contrast to e.g. Java or C++, which are *compiled*).

- widely used, especially in Data Science, ML, Computational Linguistics/NLP
- developed in the 90s, Guido van Roussum in Amsterdam
- dynamically typed: types are determined automatically at runtime
- the type of a variable can change
- you can check the type of variables with `type(var)`
- space matters: type-setting (tabulator or spaces have a meaning), in contrast to other programming languages

The Python REPL

The REPL (Read, Eval, Print Loop) is Python's interactive language shell. You can invoke it on the shell like this:

```
$ python
>>> x = 'world'
>>> print('hello', x)
hello world
>>> x = 3 - 3 * 6 + 2
>>> x
-13
>>> x = 'a' * 10
>>> x
'aaaaaaaaaa'
>>> quit()
```

Variables

- storage units
- content can be changed (i.e, is variable)
- variables consist of: **a data type**, a **variable name** and a variable **value**
- assignment of value to variable: `var_name = value`, e.g., `num = 17`, automatically determines the variable type (here: `int`)

```
# Python:
num = 17

# Java:
int num = 17;
```

Data Types

Object type	Example creation
Numbers (int, float)	`123, 3.14`
Strings	`"this class is cool"`
Lists (incl. nested)	`[1, 2, [1, 2]]`
Dictionaries	`{"1": "abc", "2": "def"}`
Tuples	`(1, "Test", 2)`
Files	`open("file.txt"), open("file.txt", "w")`
Sets	`set('a', 'b', 'c')`
Others	`boolean, None`
Program unit types	`functions, modules, classes`

Number data types

- Integers, floating-point numbers
- Numbers support the basic mathematical operations, e.g.:
  - + addition, - subtraction
  - * , /, // multiplication, division

```
>>> 1/4 # Floating number division
0.25
>>> 1//4 # Integer division
0
```

  - ** exponentiation

```
>>> 10**2
100
```

  - < , > , <= , >= comparison
  - != , == (in)equality

String data types

- Textual data in Python is handled with str objects, or strings.
- Immutable sequence of single characters

```
s1="first line\nsecond line" # double quotes
s2=r"first line\nstill first line" # raw string
s3="""first line
second line""". # triple quotes (may span multiple lines)
s4='using different quotes'  # single quotes
# Single quotes: 'allows embedded "double" quotes'
```

- What does the `r` in front of the opening quote in example s2 do (raw string)?
- See for more info: `https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str`

Unicode strings

Strings are Unicode UTF-8 by default in Python3:

```
>>> x = 'B\u00e4ume'
>>> print(x)
Bäume
```

Character Encodings

- A computer cannot directly store strings $\Rightarrow$ it works with bytes
- Convert a letter to something a computer can store $\Rightarrow$ character encodings
- Unicode $\Rightarrow$ attempt to assign a number to every human character
  - 149813 characters (as of version 15.1, released in 12.09.2023)
- In Python each string is a sequence of unicode code points

Character	Unicode Decimal Number	Character	Unicode Decimal Number
A	65	<space>	32
B	66	α	945
C	67	β	946
a	97	©	169
b	98	€	8364
c	99	🌎	127757

Table: Examples of characters and their Unicode decimal numbers

Character Encodings

- How to convert the Unicode numbers to bytes ⇒ UTF (Universal Character Set Transformation Format)
- Different versions, with different number of bytes to represent a single character ⇒ UTF-8, UTF-16, ...
- UTF-8 ⇒ represents characters in one, two, three or four bytes
- UTF-16 ⇒ represents characters in two or four bytes
- UTF-8 is more space efficient than UTF-16 in most cases.

Character	UTF-8 Binary	UTF-16 Binary
<space>	00100000	00000000 00100000
A	01000001	00000000 01000001
	11110000 10011111 10001100 10001101	11011000 00111100 11011111 00001101

Table: Bit Representation of Characters in UTF-8 and UTF-16

String operations I - What is the output?

```
s1 = 'the'
```

Operation	Description	Output
`len(s1)`	length of the string	
`s1[0]`	indexing, 0-based	
`s1[-1]`	backwards indexing	
`s1[0:3]`	slicing, extracts a substring	
`s1[:2]`	slicing, extracts a substring	
`s1 + ' sun'`	concatenation	
`s1 * 3`	repetition	

String operations I - Answers

```
s1 = 'the'
```

Operation	Description	Output
`len(s1)`	length of the string	`3`
`s1[0]`	indexing, 0-based	`'t'`
`s1[-1]`	backwards indexing	`'e'`
`s1[0:3]`	slicing, extracts a substring	`'the'`
`s1[:2]`	slicing, extracts a substring	`'th'`
`s1 + ' sun'`	concatenation	`'the sun'`
`s1 * 3`	repetition	`'thethethe'`

String operations II

```
s1 = 'these'
```

Operation	Description	Output
`'-'.join(s1)`	concatenate (delimiter: '-')	`'t-h-e-s-e'`
`s1.find('se')`	finds start of substring	`3`
`s1.replace('ese', 'at')`	replace substrings	`'that'`
`s1.split('s')`	splits at string	`['the','e']`
`s1.upper()`	upper case	`'THESE'`
`s1.lower()`	lower case	`'these'`

String operations - Recap

```
a = 'Anna'
```

Operation	Output
`a[::-1]`	?
`'@'.join(a)`	?
`a[0:1]+"dmin"`	?
`a.find("a")`	?
`a.split()`	?
`"b@org.com".split("@")`	?

String operations - Solutions

```
a = 'Anna'
```

Operation	Output
`a[::-1]`	`annA`
`'@'.join(a)`	`'A@n@n@a'`
`a[0:1]+"dmin"`	`'Admin'`
`a.find("a")`	`3`
`a.split()`	`['Anna']`
`"b@org.com".split("@")`	`['b', 'org.com']`

Lists

- collection of arbitrarily typed objects
- mutable
- ordered
- no pre-determined fixed size
- initialization: `l = [123, 'spam', 1.23]`
- empty list: `l = []`

List operations I

```
l = [123, 'spam', 1.23]
```

Operation	Description	Output
`len(l)`	length of the list	`3`
`l[1]`	indexing, 0-based	`'spam'`
`l[-1]`	backwards indexing	`1.23`
`l[0:2]`	slicing, extracts a sublist	`[123, 'spam']`
`l + [4, 5, 6]`	concatenation	`[123, 'spam', 1.23, 4, 5, 6]`
`l * 2`	repetition	`[123, 'spam', 1.23, 123, 'spam', 1.23]`

List operations II

```
l = [123, 'spam', 1.23]
```

Operation	Description	Output
`l.append('NI')`	append to the end	`[123, 'spam', 1.23, 'NI']`
`l.pop(2)`	remove item	`[123, 'spam']`
`l.insert(0, 'aa')`	insert item at index	`['aa', 123, 'spam', 1.23]`
`l.remove(123)`	remove given item	`['spam', 1.23]`
`l.reverse()`	reverse list (in place)	`[1.23, 'spam', 123]`
`l.sort()`	sort list (in place)	`[1.23, 123, 'spam']`

Nested lists

Let us consider the 3x3 matrix of numbers:
`M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` . `M` is a list of 3 objects, which are lists as well and can be referred to as rows.

- `M[1]` – returns the second element in the main list: `[4, 5, 6]`
- `M[1][2]` – returns the third object situated in the second element of the main list: `6`

Dictionaries

- Dictionaries are **mappings**, not sequences
- They represent a collection of key:value pairs
- Example:
  ```
  d = {'food':'Spam', 'quantity':4, 'color':'pink'}
  ```
- Efficient access ($\sim$ constant time):
  what is the value associated with a key?
- They are mutable like lists:
  Key-value pairs can be added, changed, and removed
- Keys need to be unique and immutable – why?

Dictionary operations I

```
>>> d = {'food':'Spam', 'quantity':4, 'color':'pink'}
>>> d['food']
#Fetch value of key 'food'
'Spam'
>>> d['quantity'] += 1 #Add 1 to the value of 'quantity'
>>> d
d = {'food':'Spam', 'quantity':5, 'color':'pink'}
```

Dictionary operations I

```
>>> d = {'food':'Spam', 'quantity':4, 'color':'pink'}
>>> d['food']
#Fetch value of key 'food'
'Spam'
>>> d['quantity'] += 1 #Add 1 to the value of 'quantity'
>>> d
d = {'food':'Spam', 'quantity':5, 'color':'pink'}
>>> d['num'] += 1
????
```

Dictionary operations I

```
>>> d = {'food':'Spam', 'quantity':4, 'color':'pink'}
>>> d['food']
#Fetch value of key 'food'
'Spam'
>>> d['quantity'] += 1 #Add 1 to the value of 'quantity'
>>> d
d = {'food':'Spam', 'quantity':5, 'color':'pink'}
>>> d['num'] += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'num'
```

Dictionary operations II

```
>>> d = {}
>>> d['name'] = 'Bob'
>>> #Create keys by assignment
>>> d['job'] = 'researcher'
>>> d['age'] = 40
>>> d
d = {'name':'Bob', 'job':'researcher', 'age':40}
>>> print(d['name'])
Bob
```

Dictionary operations III

```
>>> d = {'name': 'Bob'}
>>> d2 = {'age': 40, 'job': 'researcher'}
>>> d.update(d2)
>>> d
{'job': 'researcher', 'age': 40, 'name': 'Bob'}
>>> d.get('job')
'researcher'
>>> d.pop('age')
40
>>> d
{'job': 'researcher', 'name': 'Bob'}
```

Dictionary operations IV

```
>>> d = {'name': 'Bob', 'age': 40, 'job': 'researcher'}
>>> d
{'job': 'researcher', 'age': 40, 'name': 'Bob'}
>>> d['job']
'researcher'
# alternative to pop: >>> d.pop('age')
>>> del d['age']
>>> d
{'name': 'bob', 'job': 'researcher'}
```

Dictionary operations V

```
>>> d = {'name': 'Bob', 'age': 40, 'job': 'researcher'}
>>> d
{'job': 'researcher', 'age': 40, 'name': 'Bob'}
>>> d['gender'] = "male"
>>> d
{'job': 'researcher', 'age': 40, 'name': 'Bob',
'gender': 'male'}
```

Dictionary operations - Alternatives

```
>>> #Alternative construction techniques:
>>> d = dict(name='Bob', age=40)
>>> d = dict([('name', 'Bob'), ('age', 40)])
>>> d = dict(zip(['name', 'age'], ['Bob', 40]))
>>> d
{'age': 40, 'name': 'Bob'}
>>> #Check membership of a key
>>> 'age' in d
True
>>> d.keys()
#Get keys
['age', 'name']
>>> d.values() #Get values
[40, 'Bob']
>>> d.items() #Get all keys and values
[('age', 40), ('name', 'Bob')]
>>> len(d)
#Number of entries
2
```

Tuples

- Tuples are like lists but **immutable** (like strings)
- Used to represent fixed collections of items

```
>>> t = (1, 2, 3, 4) #A 4-item tuple
>>> len(t) #Length
4
>>> t + (5, 6) #Concatenation
(1, 2, 3, 4, 5, 6)
>>> t[0] #Indexing, slicing and more
1
>>> len(t)
???
```

Sets

- Mutable
- Unordered collections of **unique** and **immutable** objects
- Efficient check ($\sim$ constant time), whether object is contained in set.

```
>>> set([1, 2, 3, 4, 3])
{1, 2, 3, 4}
>>> set('spaam')
{'a', 'p', 's', 'm'}
>>> {1, 2, 3, 4}
{1, 2, 3, 4}
>>> S = {'s', 'p', 'a', 'm'}
>>> S.add('element')
>>> S
{'a', 'p', 's', 'm', 'element'}
```

Sets

```
>>> s1 = set(['s', 'p', 'a', 'm', 'element'])
>>> 'element' in s1
True
>>> 'spam' in s1
False
>>> s2 = set('ham')
>>> s1.intersection(s2)
{'m', 'a'}
>>> s1.union(s2)
{'s', 'm', 'h', 'element', 'p', 'a'}
```

$\Rightarrow$ intersection and union return a new set, the original sets stay unchanged

Immutable vs. Mutable

- Immutable:
  - numbers
  - strings
  - tuples
- Mutable:
  - lists
  - dictionaries
  - sets
  - newly coded objects

Control flow: if-statements

```
>>> if x < 10:
        print("too low")
...
>>> if color == "red":
        print("You selected red")
    else:
        print("Choose a color")
```

Control flow: if-statements

```
>>> x = 'killer rabbit'
... if x == 'roger':
...     print('a shave and a haircut')
... elif x == 'bugs':
...     print('whats up?')
... else:
...     print('run away!')
run away!
```

Note!

The `elif` statement is the equivalent of `else if` in Java or `elsif` in Perl.

Control flow: While loops

```
>>> while True:
...     print('Type Ctrl-C to stop me!')
...
>>> x = 'spam'
... while x: #while x is not empty
...     print(x)
...     x = x[1:]
...
spam
pam
am
m
```

$\Rightarrow$ `x[len(x):len(x)]` returns the empty string.

Control flow: For loops

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable objects (strings, lists, tuples, and other built-in iterables, as well as new user-defined iterables).

```
l = [1, 2, 3, 4]
for i in l:
    print(i)
for i in range(0, len(l)):
    print(i)
for i in range(0, 5):
    print(i)
for i in range(0, 5)[1:-1]:
    print(i)
```

Files: Read file line by line

```
# the 'with' notation makes sure that the file
# is properly closed at the end
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read & print the entire file
>>>     print(reader.read())
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
```

Further reading:
https://realpython.com/read-write-files-python/

Files: Write file line by line

```
file_name = 'outfile.txt'
lines = ['line1', 'second line', 'another line', 'last one']

with open(file_name, mode='w') as f:
    for line in lines:
            f.write(line + '\n')

#The default input/output encodings are dependent on the system's
# locale settings. You should set it explicitly in the call to
#  open(...).
# >>> with open('x.txt', 'w', encoding='utf-8') as f:
```

Data Types - What we cover today (recap)

Object type	Example creation
Numbers (int, float)	`123, 3.14`
Strings	`"this class is cool"`
Lists (incl. nested)	`[1, 2, [1, 2]]`
Dictionaries	`{"1": "abc", "2": "def"}`
Tuples	`(1, "Test", 2)`
Files	`open("file.txt"), open("file.txt", "w")`
Sets	`set('a', 'b', 'c')`
Others	`boolean, None`
Program unit types	`functions, modules, classes`

Functions

- A function is a device that groups a set of statements so they can be run more than once in a program
- Why use functions?
  - maximizes code reuse and minimizes redundancy
  - enables procedural decomposition
  - enables recursive programming and no multiply-nested loops

Defining functions

Functions are defined using the `def` keyword. They can have zero or more arguments and may return a result.

```
def function_name(arg1, arg2, ..., argN):
    statements...
    return result
```

To call a function use `function_name(arg1, arg2, ...)`.

```
>>> print('today') # calling build in function print
today
```

Function objects

Functions are normal python objects – they can be bound to other variables, be put into lists or dictionaries and even be used as parameter for other functions.

```
>>> def add(x, y):
...     return x + y
>>> def mul(x, y):
...     return x * y
>>> def do(x, y, f):
...     return f(x, y)
>>> f = mul # Bind the variable f to the function mul.
>>> f(3, 5)
15
>>> f = add # Bind the variable f to the function add.
>>> do(8, 4, f)
12
```

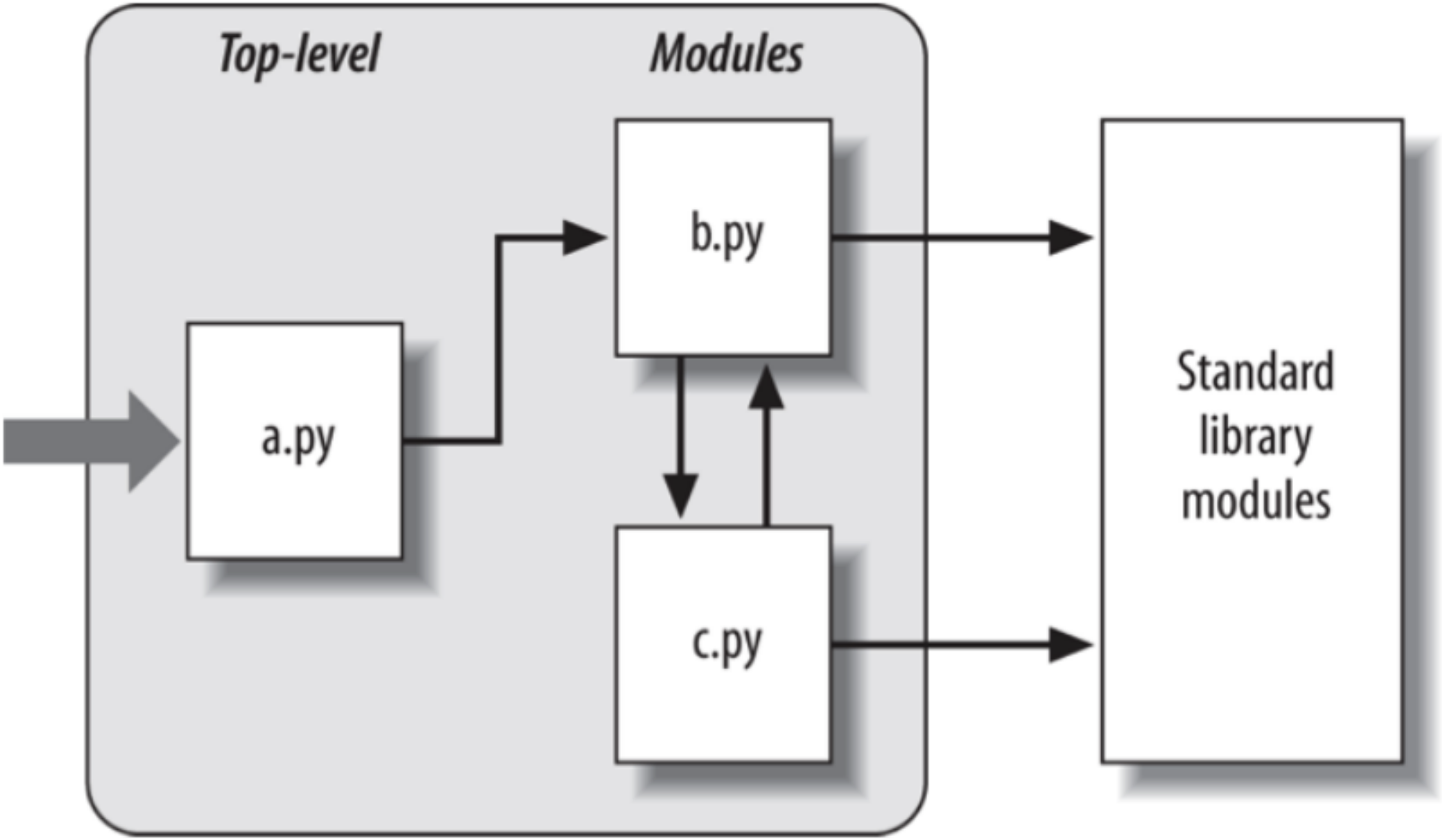
Function objects II

```
>>> # Put the functions into a dictionary.
>>> d = {}
>>> d['addiere'] = add
>>> d['multipliziere'] = mul
>>> print(d)
{'addiere': <function add at 0x7f21432cd8b0>,
  'multipliziere': <function mul at 0x7f214472f670>}
>>> # Call a function using its names in the dictionary.
>>> d['multipliziere'](3, 14)
42
```

Modules

- Packaging of program code and data for reuse
- Provides self contained namespaces that avoid variable name clashes across programs
- The names that live in a module are called its attributes
- one Python file $\sim$ one module
- Some modules provide access to functionality written in external languages such C++ or Java (wrappers)

Module imports



Modules

- import: lets a client (importer) fetch a module as a whole
- from: allows clients to fetch particular names from a module
- as: lets you rename imported names

```
>>> import collections #imports entire module
# import specific part of module
>>> from collections import defaultdict
# exercise to check what this module does :)
>>> d = defaultdict(int)
>>> import numpy as np  # as short name
```

Installing modules locally in a virtual environment

```
# Create the virtual environment in the `env` folder.
# Very useful to have diverse Python environments
$ python3 -m venv env
$ source env/bin/activate
$ pip install nltk
$ python3
>>> import nltk
>>> from nltk.corpus import stopwords
>>> nltk.corpus.download('stopwords')
>>> stopwords.words('english')
['i', 'me', 'my', ...]
>>> quit()
# Deactivate the virtual environment.
$ deactivate
```

Summary

- Data types: numbers, strings, tuples, lists, dictionaries
- Mutable / Immutable
- If-statement, while-loop, for-loop
- Reading / writing from files
- Functions
- Importing modules

Any questions?
Next: Python Tests

Outline

1. doctest Module
2. unittest Module

doctest

Python Testing: Motivation

- It is unavoidable to have errors in code.
- Unit-testing helps you ...
  - ... to catch certain errors that are easy to automatically detect.
  - ... to be more clear about the specification of the intended functionality.
  - ... to be more stress-free when developing.
  - ... to check that functionality does not change when you re-organize or optimize code.
- Today, we will look at two frameworks for unit testing that come prepackaged with Python
  1. `doctest`: A simple testing framework, where example function calls (together with their expected output) are written into the docstring documentation, and then are automatically checked.
  2. `unittest`: A framework, where several tests can be grouped together, and that allows for more complex test cases.

Simple Tests: the doctest module

- Searches for pieces of text that look like interactive example Python sessions inside of the **documentation parts** of a module.
- These examples are run and the results are compared against the expected value.
- example_module.py

```
def square(x):
    """
    Return the square of x.
    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x
```

Running the tests

```
$ python3 -m doctest -v example_module.py
Trying:
    square(2)
Expecting:
    4
ok
Trying:
    square(-2)
Expecting:
    4
ok
1 items had no tests:
    example_module
1 items passed all tests:
   2 tests in example_module.square
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
$
```

Test-Driven Development (TDD)

- Write tests first (implement functionality later)
- Add to each test an empty implementation of the function (use the `pass`-statement)
- The tests initially all fail
- Then implement, one by one, the desired functionality
- Advantages:
  - Define in advance what the expected input and outputs are
  - Also think about important boundary cases (e.g. empty strings, empty sets, `float(inf)`, 0, unexpected inputs, negative numbers)
  - Gives you a measure of progress (*"65% of the functionality is implemented"*) - this can be very motivating and useful!

TDD: Initial empty implementation

- example_module.py

```
def square(x):
    """
    Return the square of x.
    >>> square(2)
    4
    >>> square(-2)
    4
    """
    pass
```

Initially the tests fail

```
$ python3 -m doctest -v example_module.py
Trying:
    square(2)
Expecting:
    4
**********************************************************************
File "/home/ben/tmp/example_module.py", line 4, in example_module.square
Failed example:
    square(2)
Expected:
    4
Got nothing
Trying:
    square(-2)
Expecting:
    4
**********************************************************************
File "/home/ben/tmp/example_module.py", line 6, in example_module.square
Failed example:
    square(-2)
Expected:
    4
Got nothing
1 items had no tests:
    example_module
**********************************************************************
1 items had failures:
   2 of   2 in example_module.square
2 tests in 2 items.
0 passed and 2 failed.
***Test Failed*** 2 failures.
$
```

unittest

The `unittest` module

- Similar to Java's *JUnit* framework.
- Most obvious difference to `doctest`: test cases are not defined inside of the module which has to be tested, but in a separate module just for testing.
- In that module ...
  - `import unittest`
  - import the functionality you want to test
  - define a class that inherits from `unittest.TestCase`
    - ★ This class can be arbitrarily named, but `XyzTest` is standard, where `Xyz` is the name of the module to test.
    - ★ In `XyzTest`, write member functions that start with the prefix `test...`
    - ★ These member functions are automatically detected by the framework as tests.
    - ★ The tests functions contain `assert`-statements
    - ★ Use the `assert`-functions that are inherited from `unittest.TestCase` (do not use the Python built-in `assert` here)

Different types of asserts

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Question: ... what is the difference between "`a == b`" and "`a is b`"?

Example: using unittest

- test_square.py

```
1  import unittest
2  from example_module import square
3
4  class SquareTest(unittest.TestCase):
5      def testCalculation(self):
6          self.assertEqual(square(0), 0)
7          self.assertEqual(square(-1), 1)
8          self.assertEqual(square(2), 4)
```

Example: running the tests initially

- test_square.py

```
$ python3 -m unittest -v test_square.py
testCalculation (test_square.SquareTest) ... FAIL

======================================================================
FAIL: testCalculation (test_square.SquareTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/ben/tmp/test_square.py", line 6, in testCalculation
    self.assertEqual(square(0), 0)
AssertionError: None != 0

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
$
```

Example: running the tests with implemented functionality

```
$ python3 -m unittest -v test_square.py
testCalculation (test_square.SquareTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
$
```

SetUp and Teardown

- if implemented, `setUp` and `teardown` are recognized and executed automatically before and after the unit test are run, respectively
- `setUp`: Establish pre-conditions that hold for several tests.
  Examples:
  - Prepare inputs and outputs
  - Establish network connection
  - Read in data from file
- `tearDown` (less frequently used): Code that must be executed after tests finished
  Example: Close network connection

Example using setUp and tearDown

```
class SquareTest(unittest.TestCase):
    def setUp(self):
        self.inputs_outputs = [(0,0),(-1,1),(2,4)]

    def testCalculation(self):
        for i,o in self.inputs_outputs:
            self.assertEqual(square(i),o)

    def tearDown(self):
        # Just as an example.
        self.inputs_outputs = None
```

Summary

- Test-driven development
- Using `doctest` module
- Using `unittest` module
- Also have a look at the online documentation:
  `https://docs.python.org/3/library/unittest.html`
  `https://docs.python.org/3/library/doctest.html`

Any questions?

Next week: Python Classes and Objects, Python Fundamentals Part II

Appendix

Unicode strings

Strings are Unicode UTF-8 by default in Python3:

```
>>> x = 'B\u00e4ume'
>>> print(x)
Bäume
```

Let's convert from UTF-8 to binary:

```
>>> y = x.encode('utf-8') # same as x.encode()
>>> print(y)
b'B\xc3\xa4ume'
```

- \x tells Python to interpret the next two characters as hex values.
- Python displays the first 127 characters (ASCII) in their plain form (`'B'`, `'u'`, `'m'`, `'e'`). However, their internal binary representations are `'\x42'`, `'\x75'`, `'\x6d'` and `'\x65'`.

Character Encodings

```
>>> import sys    # imports the module
>>> sys.stdout.encoding
'utf-8'
>>> # alternatively:
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'

>>> # get unicode code point for a letter in decimal
>>> ord('a')
97
>>> # convert back to character
>>> chr(97)
'a'
```

UTF-8 Encoding

Example continued...

```
# Decode binary back to string:
>>> str(b"\x42\xc3\xa4\x75\x6d\x65", "utf-8")
'Bäume'
```

How does conversion from unicode to binary work?

- Naive approach:
  - Use lookup table that maps characters (i.e. their unicode code points) to a binary code that can be stored on disk.
  - E.g., we could directly store the letter A by the binary code of it's code point `ord('A')=65`, which is `1000001` (8 bits).
  - Disadvantage: Fix code lengths are space in-efficient.
    Remember: storing the symbol 🌎 requires 32 bits.
- UTF-8 encoding is a method that translates unicode code points into variable-length binary codes (`https://www.utf8-chartable.de/`).

Credits

Some slides are adapted from prior versions by Katerina Kalouli, Annemarie Friedrich, Benjamin Roth and Florian Fink. Slides on character encoding added by Ivo S. Bueno Júnior and Robert Litschko.