Schriftenreihe Fachbereich Informatik

2017-2

Informatik
Hauptcampus

HOCHSCHULE TRIER

Teaching Theoretical Computer Science with Python

Heinz Schmitz*

December 4, 2017

Abstract

We show how teaching classical content from Theoretical Computer Science (TCS) can be combined with an introduction to the Python programming language within the same course as part of a Bachelor program in Computer Science. Our main motivation to do so is to obtain executable versions of TCS models and to introduce Python in a thorough way for follow-up courses. Due to Python's reduced syntax and hence pseudocode-like appearance, TCS models can be represented and also executed in a concise way very close to the formal definitions. We outline the resulting course structure and show how Python can be interlaced with a typical sequence of lectures on TCS while preserving the mathematical nature of the course. We believe that this combination can help students to bridge the putative gap between theoretical and more practical aspects of CS, and that it provides a more seamless integration of TCS into the overall curriculum.

1 Introduction

Introductory courses on Theoretical Computer Science are a fundamental part of most study programs in Computer Science (CS) at German universities, and there is a broad agreement what subjects should usually be covered [ua16]. These subjects can be roughly termed as automata theory and formal languages, computability and complexity theory, and each time mathematical models serve as abstract representations of some computation resources and processes. Students are expected to develop the capability to design and analyse these models in order to understand the fundamental relations that delimit and determine the field. But also more practical learning outcomes are wanted: Formalization abilities and rigorous thinking are seen as important ingredients to high-quality work results throughout entire CS.

As has been frequently reported in the literature, students often have difficulties following in particular the TCS courses, resulting in substantially high failure rates and unsatisfactory learning outcomes. So various approaches have been suggested to improve this situation, among them strengthening the historical context [CGM04], visualizations of mathematical models [RBFR06] or a cognitive apprenticeship approach [KKB14]. The latter is a concept that addresses the whole process of teaching and learning, since there does not seem to be a single and stable set of influencing factors. We contribute to this ongoing discussion with an add-on to the content of such a course but within the bounds of mandatory TCS subjects.

Our impression is that students see TCS models mainly as some 'artificial' objects with no obvious links to practical areas of CS, although references to these other areas and to forthcoming courses are usually given in a TCS lecture. On the other hand, (small) models of programming languages are treated as models for computation which have a clear and strong connection to what students experience elsewhere in CS. Based on this simple observation, we

*Hochschule Trier, Schneidershof, D-54293 Trier, Germany, H.Schmitz@hochschule-trier.de

derive the following ideas to help students bridge the putative gap between theoretical and more practical aspects of CS:

- We use a small subset of a widespread programming language as a WHILE language when teaching computability. This has been common practice in TCS courses before, e.g., RIES as a PASCAL-like language in [Wag03]. Here we choose PYTHON[1] to make use of it's reduced syntax and hence pseudocode-like appearence. Together with an almost math-like notation for expressions and sets this makes PYTHON a good candidate for TCS purposes. Moreover, it can be expected that PYTHON plays it's role elsewhere in the CS curriculum.

- After that, we use this WHILE language (with some extensions) consequently to represent and execute all forthcoming TCS models in the course. This also includes implementations of algorithms derived from all constructive proofs. So on one hand the (extended) WHILE language is immediately applied to some 'real' computation tasks, and on the other hand students can investigate all TCS models and proofs by executing them in a present-day language.

- Parallel to the treatment of different TCS models, the WHILE language is stepwise extended by basic data types and their operations in order to allow more convenient programming. Also runtime analysis, first introduced for the plain WHILE language, is carried along these extentions by expanding the cost model and definitions of input lengths accordingly. So the TCS models and algorithms in PYTHON can be immediately judged by their efficiency by both, theoretical bounds and practical experiences. As a consequence, the introductory TCS course in not only a basis for CS in general and for advanced TCS courses in particular, but it can also serve as a fundament for follow-up courses on algorithms when based on PYTHON.

We believe that these elements result in a more seamless integration of TCS into the overall CS curriculum, and hopefully less students look at it as something that they just have to get along with.

The course we outline in the next section was develped over the last decade and has some accompanying features. It is given in a series of 15 lectures each 90 minutes per week, accompanied by weekly exercise sessions of the same duration with up to 30 students where homeworks need to be presented. All lectures are recorded as *screencasts* and students are asked to work through them before the weekly *tutorial* which is a session given by the instructor instead of the lecture. Here questions can be asked, additional examples and live-programming are presented and solution hints to more complicated exersices are discussed. An online *quiz* with about 500 true/false-questions about the lecture contents is intended to activate students. The main lecture material is a script where proofs and examples are added by handwritten tablet input during each lecture. Students are invited to turn it into their *personal workbook* by adding their own comments.

2 Course Outline

The content can be roughly sectioned into basics (lectures 1 to 3), computability (4,5), runtime analysis (6,7), finite automata and regular languages (8 to 11), contextfree languages (12,13) and complexity (14,15). There is no particular lecture on PYTHON since language elements are introduced en passant when needed. The teaching order is a result of the ideas mentioned before.

[1] www.python.org

The early treatment of computability together with the introduction of a PYTHON subset as a WHILE language is needed to provide a model of computation that can be immediately applied in subsequent lectures. For the same reason runtime analysis of these WHILE programs and their extensions are considered next, so this machinery can also be applied to the algorithms presented later in the course. On the other hand, complexity classes are only introduced towards the end of the course when various programming experiences have been made in previous exercises. Next we give some keywords for the content of each lecture and show how PYTHON is interlaced. We also mention some exercises that can be assigned in addition to usual TCS paperwork homeworks.

1. *Introduction.* Syllabus, organizational notes; formulation of theorems and basic proof forms.

   The PYTHON part here is only to install all needed software and get an IDE running (we use PYTHON 3.x, Eclipse[2] and the PyDev[3] plugin).

2. *Induction.* Induction on $\mathbb{N}$, inductive definitions, structural induction, examples of recursive functions.

   Additionally, we give an informal introduction to the PYTHON shell and we declare and execute a simple recursive function. It's correctness proof serves as an example for the application of the induction principle in a CS context. Moreover, the set of feasible arithmetic expressions as used later in the WHILE language are among the examples for inductive definitions, and some easy properties of these expressions (e.g. numbers of opening and closing brackets coincide) are shown by structural induction.

3. *Words and Formal Languages.* Alphabets, words, formal langauges, word problem, basic language operations.

   In this lecture, we also introduce PYTHON string and set types together with basic operations for both of them. So finite sets of words can be explicitely handled, and differences between TCS and PYTHON notations are shown, e.g.

	TCS notation	PYTHON
words, empty word	$w = 01101$, ε	`w = '01101', ''`
alphabets	$\Sigma_1 = \{0,1\}, \Sigma_2 = \{a,b,c\}$	global
length, concatenation	$\lvert w\rvert$, wv, w^5	`len(w)`, `w + v`, `w*5`
indexing, subwords	$w = a_0a_1a_2$, $a_i \cdots a_j$	`w[0]w[1]w[2]`, `w[i:j]`
sets, empty set	$A = \{\varepsilon, 0, 00\}$, $E = \emptyset$	`A = {'','0','00'}`, `E = set()`
membership	$x \in A, x \notin A$	`x in A, x not in A`
language operations	$A \cup B, A \cap B$	`A | B, A & B`
	$\overline{L}$, L_1L_2, L^k, L^*, L^R	no such thing

   Additional exercises cover implementations of language operations in PYTHON where finiteness is preserved.

4. WHILE *Programs.* Models of computation, syntax and semantics of a WHILE language, examples.

   We introduce a carefully chosen subset of PYTHON as a model for computation so students immediately have the PYTHON shell as an execution environment available. The syntactic elements are inductively introduced together with their semantics and comprise:

[2] http://www.eclipse.org
[3] http://www.pydev.org

Constants $[-](0|(1|\ldots|9)(0|\ldots|9)^*)$

Identifiers $(a|\ldots|Z)(a|\ldots|Z|0|\ldots|9)^*$

Expressions constants, variables
`(a+b)`, `(a-b)`
`f(b1,...,bm)`

Conditions `(a==b)`, `(a!=b)`, `(a>b)`, `(a<b)`, `(a>=b)`, `(a<=b)`, `(not c)`, `(c1 or c2)`, `(c1 and c2)`

Assignment `a = b`

Cond.Stmts
```
if c:
    s1
else:
    s2
```
```
if c:
    s
```

Composition
```
s1
s2
```

Loops
```
while c:
    s
for i in range(a1,a2):
    s
```

Functions
```
def f(a1,...am):
    s
    return a
```
```
def f(a1,...am):
    return a
```

Programs
```
f1
...
fm
```

Typical exercises ask for partial functions φ_P on $\mathbb{Z}$ computed by a given WHILE program P, and vice versa. For the latter, students need to provide implementations that pass at least the given test cases.

5. *Computability.* Class of WHILE-computable functions, equivalence to other models of computation (w/o proof), Turing-completeness, Church-Turing-hypothesis; existence of incomputable functions by counting (diagonalization); decidability, examples of undecidable sets.

   Additional practical exercises for this lecture treat algorithms for characteristic functions of some example problems, a WHILE implementation of the Ackermann function and (pseudo-) WHILE algorithms for closure properties of recursive sets.

6. *Runtime Analysis.* Types of cost models, uniform model for WHILE, step function, runtime function, input length, asymptotic classes and their properties, runtime analysis of WHILE programs.

   The main focus in the additional exercises is to analyse the runtime of given WHILE programs. However, students also gain hands-on experience in the significance of theoretical bounds, e.g., when experimentally comparing the naïve WHILE program for multiplication by successive summation with a more clever algorithm for it.

7. *Python Programs.* Specification of computational problems, classical examples like KNAPSACK and pattern matching in strings; extension of WHILE programs by data types `str`, `set`, `list` and `dict`, and their operations; expansion of the cost model, definition of input length for the new data types, examples of runtime analysis.

   More classical decision and optimisation problems are introduced in the exercises, e.g., BIN PACKING and SOS. Exhaustive search algorithms and their runtime analysis using the new data types need to be derived.

8. *Deterministic Finite Automata.* Examples and definition of DFAs, extended transition function and accepted language; decision problems for DFAs; extended example of search automata for string matching.

Together with the formal definitions, DFAs are immediately represented in PYTHON. All ingredients needed to do so have been introduced before. We contrast an example DFA with its PYTHON representation.

state set $Q = \{q_0, q_1, q_2\}$
alphabet $\Sigma = \{0, 1\}$
accepting states $F = \{q_2\}$
transition function δ as:

q	a	$\delta(q, a)$
q_0	0	q_1
q_0	1	q_0
q_1	0	q_1
q_1	1	q_2
q_2	0	q_2
q_2	1	q_2

DFA $A = (Q, \Sigma, \delta, q_0, F)$

```
Q = {0, 1, 2}
Sigma = {'0', '1'}
F = {2}
delta = {}

delta[0,'0'] = 1
delta[0,'1'] = 0
delta[1,'0'] = 1
delta[1,'1'] = 2
delta[2,'0'] = 2
delta[2,'1'] = 2

A = [Q, Sigma, delta, 0, F]
```

Similarly, the inductive definition of the extended transition function has an immediate recursive implementation in PYTHON.

ext. trans. function $\widehat{\delta} : Q \times \Sigma^* \to Q$ as

- $\widehat{\delta}(q, \varepsilon) = q$, and
- $\widehat{\delta}(q, wa) = \delta(p, a)$ if $p = \widehat{\delta}(q, w)$

```
def delta_hat(delta, q, v):
    if v  == '': return q
    w, a = v[:-1], v[-1]
    p =  delta_hat(delta, q, w)
    return delta[p, a]
```

Finally, the definition of the accepted language has an executable counterpart in PYTHON.

$L(A) = \{w \in \Sigma^* \mid \widehat{\delta}(q_0, w) \in F\}$

```
def run_dfa(A, w):
    [Q, Sigma, delta, q0, F] = A
    return delta_hat(delta, q0, w) in F
```

Observe that no more code is needed to run a DFA on some input word. Students benefit from this in the exercises when the behaviour of self-designed automata for given languages can be inspected. Moreover, algorithms for emptiness and finiteness tests for a given DFA can be programmed in PYTHON. An example of a more advanced task is to implement the construction of the search automata for string matching as treated in the lecture.

9. *Nondeterministic Finite Automata.* Examples and definition of NFAs, extended transition function and accepted language; equivalence of DFAs and NFAs, closure properties.

   Again, NFAs are represented in PYTHON and also all other definitions are carried over. Similarly few code is needed to obtain executable NFAs. The runtime analysis of the respective `run_nfa` function reveals the price to pay for less states and easier design. Evident exercises cover implementations for the constructive proofs given by the powerset construction and for the closure properties.

10. *Regular Expressions.* Syntax and semantics of regular expressions, conversion to NFAs, construction of a regular expression for given DFA with a Floyd/Warshall-like algorithm.

Although regular expressions are treated only as flat strings in PYTHON, they will be parsed in later exercises. Here regular expressions are used to specify valid WHILE constants and identifiers, and automata need to be obtained to recognize them by the methods introduced in the lecture. Additionally students are asked to implement the construction of the regular expression from a given DFA by dynamic programming. Here they gain first-hand experience about what an exponential growth of the output size looks like.

11. *Regular Languages.* Proof and applications of Pumping Lemma. Summary of lectures 8 to 11.

    Students have implemented and experienced up to now a collection of algorithms transforming different representations of regular languages. This own experience adds to the motivation of asking for which languages these nice things are not possible. The summary additionally emphasizes the derived runtimes of simulation and transformation algorithms, and reveals trade-offs between regular language representations.

12. *Contextfree Languages.* Definition and examples of contextfree grammars, derivations, from DFAs to CFGs; PYTHON data type `tuple`, equivalence of left-, right-derivations and parse trees, ambiguity; ε-free grammars and their construction.

    The representation of CFGs in PYTHON is again chosen close to the formal definitions, where each production is just a tuple of the left and right side. This may also apply for other types of grammars.

terminals $\Sigma = \{0, 1\}$
non-terminals $N = \{S\}$
set of productions P:
$S \rightarrow \varepsilon$
$S \rightarrow 0S1$
CFG $G = (\Sigma, N, P, S)$

```
Sigma = {'0', '1'}
N = {'S'}

P = {   ('S', ''),
        ('S', '0S1') }
G = [Sigma, N, P, 'S']
```

Based on this, a leftmost derivation step for a given production is easy to implement.

if $p = (A \rightarrow \beta)$

and $\alpha = wA\gamma$

$\alpha \Rightarrow_l w\beta\gamma$

```
def derive_left(alpha, p):
    (A, beta) = p
    if A in alpha:
        i = alpha.index(A)
        w, gamma = alpha[:i], alpha[i+1:]
        return w + beta + gamma
    else:
        return alpha
```

Additional exercises cover the implementation of turning a PYTHON representation of an arbitrary DFA into the respective CFG, or to return the set of all words that can be produced with at most k leftmost derivation steps for a given CFG.

13. *An All-round Parser.* Chomsky normal form, CYK algorithm, correctness and runtime; parsing arithmetic expressions of WHILE programs.

    The lecture demonstrates the small step from the inductive definition of subproblems for the CYK method to an executable implementation by dynamic programming. Additional exercises ask for parsing WHILE expressions including function calls, where results from

previous exercises on valid WHILE identifiers need to be reused, and for parsing regular expressions over a given alphabet.

14. *The Classes P and NP.* Definition of P, FP and EXP, closure properties of P, common problem structure of SOS, TSP and others; definition of NP, inclusions from P and to EXP.

    Here NP is defined via length-bounded certificates and poltime verification, so in additional practical exercises students can implement these verifications for given solution candidates and convice themselves of their poltime by runtime analysis. Moreover, suitable iterators are available in PYTHON's `itertools` in order to obtain canonical exhaustive search algorithms in a straightforward way, while runtime analysis of these PYTHON implementations shows membership in EXP. Here is the entire algorithm for the SOS problem.

    for $a = (a_0, \ldots, a_{m-1}), k$

    exists $I \subseteq \{0, \ldots, m-1\}$ with $k = \sum_{i \in I} a_i$?

```
from itertools import product
def sos_exhaustive(a,k):
    m = len(a)
    for I in product((0,1),repeat=m):
        if k==sum(a[i] for i in range(m)
                              if I[i]):
            return 1
    return 0
```

15. *NP-Completeness.* Poltime-many-one-reducibility, example reductions, closure of P and NP under $\leq_m^p$, notion of NP-completeness, examples (w/o proof), discussion of P/NP-question.

    Typical exercises ask for reductions between similar problems extended by implementations of these reduction functions.

3 Outlook

Time is the resctrictive factor when including additional topics, and other things have to be left out. In our case, we moved the dicussion of Turing machines and proofs of their equivalence to other models to later courses. The same holds for a self-contained NP-completeness proof and the other levels of the Chomsky hierarchy. We found that adding PYTHON implementations also in these later courses is helpful there too, and not much additional effort is needed after this introductory course, e.g., to obtain a straightforward representation and execution of Turing machines or the implementation of their Gödelization. Another follow-up course that is now well-prepared may deal with syntactic analysis, where discussion and representation of PDAs can be easily motivated and combined with the development and implementation of parsing algorithms – based on PYTHON and the introduced cost model. The same is true for a course on basic algorithms that can follow seamlessly [Het10, Häb12]. The introductory TCS course described here does not only allow for additional exercises, but it also yields subjects for practical but theory-based student projects or theses that are well-prepared by this course. E.g., this could be the development of a WHILE parser that adds to each program a counter for computation steps according to the cost model, or a PYTHON module to visualize and interact with the transition diagrams of the automata models, or a collection of implementations of reduction functions that make use of some solver package for satisfiability or linear optimization in the background, only to name a few.

Acknowledgements

This teaching concept has been jointly developed and intensely discussed with Christian Glaßer, Universität Würzburg, Germany. We are also grateful to Christian Reitwießner for pointing out the advantages of PYTHON to us.

References

[CGM04] Carlos Iván Chesñevar, María Paula González, and Ana Gabriela Maguitman. Didactic strategies for promoting significant learning in formal languages and automata theory. *ITiCSE*, page 7, 2004.

[Häb12] T Häberlein. *Praktische Algorithmik mit Python.* Oldenbourg Verlag, 2012.

[Het10] M Hetland. *Python Algorithms.* Apress, 2010.

[KKB14] Maria Knobelsdorf, Christoph Kreitz, and Sebastian Böhne. Teaching theoretical computer science using a cognitive apprenticeship approach. *SIGCSE*, pages 67–72, 2014.

[RBFR06] Susan H Rodger, Bart Bressler, Thomas Finley, and Stephen Reading. Turning automata theory into a hands-on course. In *the 37th SIGCSE technical symposium*, page 379, New York, New York, USA, 2006. ACM Press.

[ua16] unknown author. *Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen (Juli 2016).* Gesellschaft für Informatik e.V., 2016.

[Wag03] K W Wagner. *Theoretische Informatik – Eine kompakte Einführung.* Springer, second edition, 2003.