

# Programming Course Cookbook

Kristian Rother

[www.rubor.de](http://www.rubor.de)

## ***Introduction***

This is a collection of my favourite recipes for teaching programming. If you are a lecturer looking for ideas to spice up your classes and make them more interesting for your students, this might be the right literature for you.

I called the individual methods and modules recipes because each of them can be applied independently, and many can be integrated easily with existing course material. Didactic theory is not covered here, and I am not giving hints for giving the standard PowerPoint-based frontal lecture – which I don't think is a too brilliant idea anyway.

This material was developed during a Python programming course for bioinformatics students at the Adam Mickiewicz University in Poznan, Poland. However, a lot of the material should work for other topics as well.

I dedicate this document to Bettina Ritter-Mamczek and Cornelius Frömmel, to whom I owe both skills and passion to teaching.

*Written in May 2007  
Last Update: December 2007*

# Contents

Introduction.....	1
The Python Course Cookbook.....	4
1. Decide what to teach.....	4
<i>Recipe 1.1: Preparing the menu (content reduction)</i> .....	4
2. Preparing a lesson.....	5
<i>Recipe 2.1: Method changes</i> .....	5
3. Warming up with your Class.....	6
<i>Recipe 3.1: Diving in</i> .....	6
<i>Recipe 3.2: Glucose Surge</i> .....	6
4. Teaching new Stuff.....	7
<i>Recipe 4.1: Kickstop</i> .....	7
<i>Recipe 4.2: New York Minute</i> .....	7
<i>Recipe 4.3: Upside-down</i> .....	8
<i>Recipe 4.4: Puzzle</i> .....	8
<i>Recipe 4.5: Mind Mapping</i> .....	9
<i>Recipe 4.6: Mind-Mapping-related Techniques</i> .....	9
<i>Recipe 4.7: Brainstorming</i> .....	9
<i>Recipe 4.8: Advance meter</i> .....	9
<i>Recipe 4.9: Tree Traversal Live</i> .....	10
<i>Recipe 4.10: Tree Selection Sort Live</i> .....	10
<i>Recipe 4.11: The Backpack Problem</i> .....	11
<i>Recipe 4.12: Anagrams</i> .....	11
5. Exercises.....	12
6. Repeating.....	12
<i>Recipe 6.1: Domino</i> .....	12
<i>Recipe 6.2: Crossword</i> .....	13
<i>Recipe 6.3: Return Home</i> .....	13
<i>Recipe 6.4: Pro and Contra</i> .....	13
<i>Recipe 6.5: A Fistful of Explanations</i> .....	13
7. Enforce good Student Presentations.....	14
<i>Recipe 7.1: Rules for presentations</i> .....	14
8. Get Feedback.....	14
<i>Recipe 8.1: Closed poll</i> .....	14
<i>Recipe 8.2: Only one cross each</i> .....	15
<i>Recipe 8.3: Burning for Python</i> .....	15
Appendix.....	15
1. Exercises on String handling.....	15
<i>Exercise 1.1: String handling using standard functions I</i> .....	15
<i>Exercise 1.2: String handling using standard functions II</i> .....	15
<i>Exercise 1.3: Printing strings</i> .....	16
<i>Exercise 1.4: Regular expressions I</i> .....	16
<i>Exercise 1.5: Regular expressions II</i> .....	16
<i>Exercise 1.6: Regular expressions III</i> .....	16
2. Exercises on Classes.....	16
<i>Exercise 2.1: An Item class</i> .....	16
<i>Exercise 2.2 An ItemList class</i> .....	16
<i>Exercise 2.3 A Backpack class</i> .....	16
3. Exercises on Reading and Writing Files.....	17
<i>Exercise 3.1: Creating random data</i> .....	17

<i>Exercise 3.2: Creating non-random data</i> .....	17
<i>Exercise 3.3: Read a tab-separated file</i> .....	17
4. Exercises on Data Structures and OOP.....	17
<i>Exercise 4.1: Write a stack class</i> .....	17
<i>Exercise 4.2: Write a queue class using iterators</i> .....	18
<i>Exercise 4.3: Write a general order tree class</i> .....	18
<i>Exercise 4.4: Write a binary tree class</i> .....	18
<i>Exercise 4.5: Traversing a tree</i> .....	19
<i>Exercise 4.6: Searching in a tree</i> .....	19
<i>Easy Exercise 4.7: Special Methods</i> .....	20
<i>Advanced Exercise 4.8: Sort a binary tree</i> .....	20
<i>Advanced Exercise 4.9: An abstract superclass</i> .....	20
<i>Supplementary Material: Tree Traversal</i> .....	21
<i>Supplementary Material: Representing a Binary Tree as an Array</i> .....	21
<i>Supplementary Material: Representing a General Ordered Tree by a Binary Tree</i> .....	22
<i>Supplementary Material: Tree Selection Sort</i> .....	23
5) Exercises on Dynamic Programming.....	23
<i>Summary: The Backpack Problem</i> .....	23
<i>Basic item list</i> .....	23
<i>Pseudocode of the dynamic programming algorithm</i> .....	24
<i>Advanced Exercise 5.1: Dynamic Programming</i> .....	24
<i>Advanced Exercise 5.2: Probing the algorithm</i> .....	24
Glossary.....	24

# ***The Python Course Cookbook***

## **1. Decide what to teach**

As a teacher, your job is to get your students to learn efficiently. You cannot force them to do so; it is always the students who decide whether they learn or not. Like you cannot force children to eat. But you decide what is on the menu, and you should make sure it looks tasty.

To a student a new, unknown topic looks at first interesting, but also big and intimidating. So the first thing you should do is to chop it up into small parts the students can digest more easily. They have a right to know what is expecting them.

### ***Recipe 1.1: Preparing the menu (content reduction)***

#### **Material:**

subject you want to teach  
pen  
a few sheets of paper

#### **Method:**

Collect all possible concepts and sub-topics revolving around your subject from the corners of your brain and write them down. Sort them into 4-7 groups that make sense to you (hierarchical clustering). There is no perfect grouping, this depends a lot on your taste. The group sizes may vary as well. Choose a proper name for each group. Draw a nice diagram containing the subject as a headline, and place the group names there.

**Time:** 20'

#### **Example:**

In my Python course, i wanted to focus on these five aspects, i chose as anchors (with the sub-subtopics in brackets):

- Structure (modules, functions, exceptions and classes)
- Operations (string handling, regular expressions, file handling, boolean expressions, iterators, and web operations).
- Data (data types and data structures)
- Algorithms (sorting and dynamic programming)
- Libraries (BioPython and others)

Of course, there are many other interesting things in Python, but they will be saved up for a different course. Once in the class, i drew the picture of a big snake, and wrote 'Algorithms' near its brain, 'Data' to the big bulge in its stomach, 'Libraries' to a nest of eggs close to its tail etc.

**(a picture might be better here)**

Once your students know that “*Programming Python consists of Structures, Operations, Data, Algorithms and Libraries.*” they have some mental boxes to drop the details into. Thus, communicating the menu should be done at the very beginning. It diminishes the subjective size of the topic, hence the name *content reduction*.

Your menu does not have to show in what order things will be treated. You might want to prepare lessons that cover two aspects of your menu at a time or start improvising at some point. It is more important to visualize your menu: Prepare a picture or clear diagram. Don't put all the sub-subtopics into it, just the group names are enough for the beginning. Try to re-use this diagram repeatedly, so the students will check what they are already familiar with.

This recipe can be applied not only to an entire course, but also on any single lesson. You can apply it recursively to your menu, to create a new structure diagrams for each of your sub-topics.

## 2. Preparing a lesson

The attention of people listening decreases over time. After 20 minutes, most of their brains normally switch off. To avoid that students get sleepy or think of something more interesting one by one, a lesson needs to alert their brains that something interesting is going on. Just talking interestingly is by far not enough. The trick is to build in some rapid turn every 20 minutes at least.

### **Recipe 2.1: Method changes**

#### **Material:**

- powerpoint presentation
- black/whiteboard presentation
- asking questions
- explanations by students
- students working on their own
- students working together
- discussion
- any other recipe

#### **Method:**

After spending up to 20 minutes with one method from the list above, change it for another one.

**Time:** 20' or less for each method

#### **Useful hints:**

- It is useful to prepare a lesson plan, a table listing the topics for a lesson, the methods you are going to use and the time you plan for this.
- The time necessary for any particular method is usually underestimated. Leave some 'buffer time' in your plan.
- If you are sharing handouts or other material to students, they will get excited and distracted from your person immediately. Make sure to tell them what they are expected to do *before* they get anything.
- A Klingon proverb says: "*Planning ends with enemy contact.*". Be ready to recognize the moment at which to trash your lesson plan.

#### ***Coming to the Classroom***

In many lecture halls, the arrangement of tables, the speakers pult, projectors, books, screens etc. form psychological barriers. They are extensively used by both students and teachers to entrench themselves. In the resulting climate, it is difficult to establish a continuous dialogue. To create a more communicative atmosphere, you should move tables aside if possible. The chairs can be arranged in a circle or half-circle. Of course this will only work if you get rid of your defences as well. Employ student workforce for rearranging furniture.

### 3. Warming up with your Class

If you spend some longer period of time with the same people, you normally want to know who they are. In a class, you should at least learn their names. Most students appreciate when they are addressed by their name. Having everybody introduce himself is an excellent opportunity to probe students' expectations and fears.

#### **Recipe 3.1: Diving in**

**Material:**

board with text:

*Who are you?*

*What do you like about programming?*

*What do you like not?*

**Method:**

Write the questions on the board. Start introducing yourself, and answer both questions with a short statement. Ask the student next to you to do the same. Go through the class in some logical order, if the students don't do it by themselves already. Listen. Don't comment on anything. Other questions, like "*What is your favorite Python module?*", "*What program are you unable to code?*" also work very well.

**Time:** 10'

#### **Recipe 3.2: Glucose Surge**

**Material:**

sweets

**Method:**

An extremely crude approach. Share candies to welcome your students. They will get more open and talkative immediately. This also helps to provide some extra energy after a long day of study. To make more sense of this approach you can get sweets wrapped in paper of different color. The colors can be used to assign the students to groups for an upcoming group exercise.

**Time:** 5'

## 4. Teaching new Stuff

There is a big number of ways to come up with new content to feed the students. Starting to talk about it is just one of them. The good thing about talking is that you can start it anytime. The bad thing is that you might end up talking to yourself if things go wrong. Below, I give some examples of what else you can try. As a rule of thumb, prefer a method where the students *do* something over such where they don't, and methods where they *see* over those where they *hear*.

### Recipe 4.1: Kickstop

**Material:**

board

**Method:**

Stop talking. Ask the students to draw a diagram of what they think is most important in the topic you just talked about. Give them a few minutes to take breath. Ask them to explain their diagram to their neighbors. Give them a few more minutes. Ask for brave volunteers to put their diagram to the board.

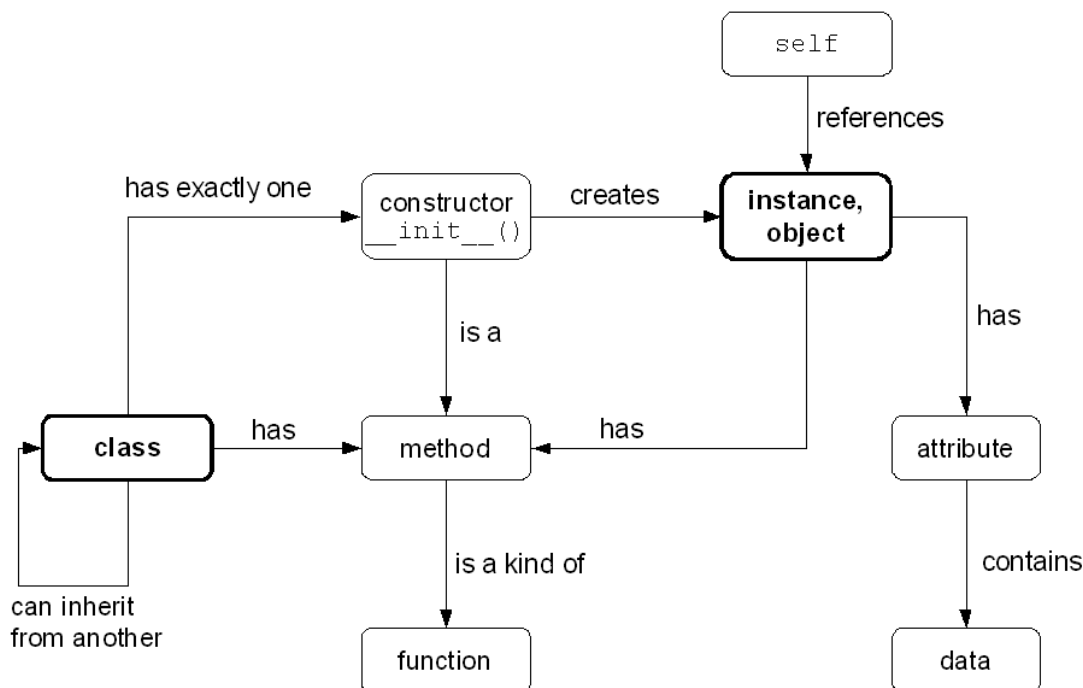
You can introduce box-and-arrow concept maps as an idea how to draw a good diagram.

**Time:** 5'

**Variation:**

Suggest a pictograph as a metaphor: “Draw a ship/spaceship/car/fish and write the parts of the program where you think they fit best. Then explain how the program works.”

### Concept Map: Object-Oriented Programming (OOP)



### Recipe 4.2: New York Minute

**Material:**

board  
half as many code examples as there are students  
(I shared eight appropriate sections from the 'Python Cookbook' revolving around lists and dictionaries).

**Method:**

Start one lesson before. Assign the students to pairs, and give each pair a code example to prepare. Ask them to explain in the next lesson 1) What it is, 2) How it works, 3) What it is good for.

In the next lesson, tell the students to explain their code within a “*New York Minute*” - a small, flexible amount of time (see Wikipedia). They may use the board. Notify and interrupt anyone who is about to talk longer than five minutes.

**Time:** 30'

**Recipe 4.3: Upside-down**

**Material:**

board

**Method:**

Write a question on the board that asks for the opposite of what you want to teach. E.g. “*How can i write a really bad program?*”, “*How can i waste memory?*”, “*Name inefficient data structures.*” etc. Let the students digest the question for 1-2 minutes and let them discuss. Then ask, and write their suggestions on the board. Add from your own notes afterwards.

**Time:** 10'

**Recipe 4.4: Puzzle**

**Material:**

board

around 20 cards in two colors (A4 sheets divided by three)

magnets or glue tape

**Method:**

Write program code lines or procedure names on one set of cards, and short clues what they do on the other. Mix the cards and attach them on the board. Let the students think for a couple of minutes. Ask one or two students (not the ones from the premium league!) to come to the board. Ask the class to assist them arranging the pairs. Listen to their discussion.

**Time:** 15'

**Example: Puzzle**

*card 1*

$1 < x < 10$

$a, b = b, a$

*card2*

range test

switching values

### **Recipe 4.5: Mind Mapping**

**Material:**

board

**Method:**

Write the topic of your lesson in the middle of the board. Ask for things connected to that. Write them on the board, connected by arrows (or without). This is a faster way to get started into a new topic than e.g. the Metaplan® technique. From a Mind Map it is easy to start a dialogue with the class.

**Time:** 10'

### **Recipe 4.6: Mind-Mapping-related Techniques**

**Material:**

board  
cards in one or more colors  
marker pens  
magnets

**Method:**

This big family of methods is for sorting a multitude of concepts in student's heads. Write a general question or concept for a topic on the board, e.g. *“What can I do with Python functions?”*, *“Data”*, *“File operations”*. Give your audience a few minutes to think. Ask them to write their ideas on the cards (in a well-visible font size). Then collect the ideas on the board and read them aloud. Arrange them into some logical way. There is a multitude of similar variants possible: connecting the words on the cards by lines, arrows, arranging them to a network, or clustering them. In the Metaplan® technique, each of the clusters gets a label on an additional card.

The results of the Mind-Mapping determine how students will categorize concepts in the subsequent lessons. It should be therefore well-prepared, and some patience is required to wait for answers. Only after that you should add things you are missing.

**Time:** 20'

### **Recipe 4.7: Brainstorming**

**Material:**

board

**Method:**

Write an understandable formulation of a new topic on the board, e.g. For the *“Data”* subject *“What data do programmers need to store?”*. Ask the students for ideas. Write everything on the board they say. Continue until they run out of ideas or the board is full. Only after that, add from your notes.

**Time:** 15'

### **Recipe 4.8: Advance meter**

**Material:**

board  
paper arrow

**Method:**

Write the outline of your presentation on the board. Attach the arrow pointing to the beginning before you start. Advance it while you go through your talk. If you have little space, replace the outline by a 0-100% scale. The advance of the meter helps your

audience to keep focused because they know how much they still have to endure.

**Time:** 1'

**Variation:**

This works in Power Point presentations as well.

#### **Recipe 4.9: Tree Traversal Live**

**Material:**

board  
small ball  
some free space

**Method:**

Display the rule set given below on the board. Ask at least seven students to arrange themselves to a binary tree in front of the class. If there is enough space, involve the entire class. Give the ball to the student resembling the root node and ask him to traverse the tree. Discuss the results, and go through traversal methods.

**Time:** 10'

##### **Instructions: tree traversal**

- If you get the ball from behind, call your name. Then give the ball to the person at your left arm
- If you get the ball from your left, give it to the person at your right arm
- If you get the ball from your right, give it to the person behind you.

#### **Recipe 4.10: Tree Selection Sort Live**

**Material:**

board  
cards with numbers from 1-7 (A5)  
some free space

**Method:**

Display the rule set given below on the board. Use the same tree arrangement as in the recipe above. Shuffle the number cards and give them to the students representing the leaf nodes. Ask the students to sort the numbers. Put someone to collect cards from the root node. Discuss the results, and start going through the theory of binary tree sorting.

**Time:** 10'

##### **Instructions: Tree Selection Sort**

- If you have a number, show it to your back side.
- If all persons in front of you are showing a number, take the smaller one.
- If you have no number, and no people in front of you, go away.

### Recipe 4.11: The Backpack Problem

**Material:**

- board
- 7-8 cards (A5)
- marker pen
- magnets or glue tape
- dynamic programming (as pseudocode)

**Method:**

Write item names, sizes and values from the table below on each card. Attach them to the board. Tell the students they are thieves and are to negotiate how to fill a backpack of size 7. Let them discuss. Then, ask for a solution. Ask how they came to this solution.

Ask how they would find the solution if they came with a van instead of a backpack next time. Get to the point of combinatorial explosion by  $2^n$ .

Write a table on the board ranging from 0 to 7 and develop the optimal backpack for each size, starting from zero. Then show them the pseudocode, and ask them to understand it.

**Time:** 15'

item	size	value	item	size	value
TV	3	200€	silver spoons	1	400€
stereo	2	1000€	jewellery	2	2000€
antic vase	3	1500€	camera	1	100€
DVD	2	600€	cash	1	3000€

### Recipe 4.12: Anagrams

*Suggested by Rob Knight*

**Material:**

- board
- dynamic programming pseudocode

**Method:**

Write a short word on the board. Ask students for anagrams (permutations of the letters). Try a longer word. Try to find out how many anagrams are there for a word of size  $n$ .

**Time:** 15'

**Example:**

WORD	DORW	ORDW	ROWD
WODR	DOWR	ORWD	RODW
WDOR	DWRO	OWDR	RDOW
WDRO	DWOR	OWRD	RDWO
WRDO	DROW	ODWR	RWOD
WROD	DRWO	ODRW	RWDO

## 5. Exercises

As a general rule, nobody from the group should be made looking like an idiot by the teacher (also called “*to make someone lose his face*”). There are different cultural conceptions on this, so use your reasoning. But you can guess that nobody likes to be subject to inquisition or criticism in front of his fellow students.

This means: *Picking people from the group at random and questioning them, nonconstructive feedback after student talks, and not interfering when students get personal among each other during the lesson are absolute don'ts!*

You should also protect the class from frustration on the level of the subject. Therefore, exercises should rather be a little too easy than too hard. When creating harder tasks, they should be formulated with an open end, if possible, so students can feel they have done well if they accomplished just 80%.

### Example that does not work well:

Implement a quicksort algorithm.

### Example task that works well:

Complete the quicksort algorithm provided by the docent. Write a program that uses this algorithm to sort different sets of data. Determine how the algorithm scales with the size of the data. Compare the performance to Python's built-in `sort()` function. Which parts of the algorithm eat up the most time?

In the appendix, I placed a few exercises on data structures and OOP. They should be performed by pairs of students. From time to time, I use a set of old memory cards to randomize groups. Of course, students are able to pair up themselves, but randomizing helps in groups of very mixed level.

## 6. Repeating

Repeating content is essential to ensure successful learning. Furthermore, giving students the opportunity to show what they have learned already motivates them. In each seminar, some time should be reserved for repeating the last lessons' material. Repeating content is essential to ensure successful learning.

### Recipe 6.1: Domino

#### Material:

cards (A4 paper cut by three). (Two cards more than students are required).  
magnets or sticky tape to attach cards on the board.

#### Method:

Write concepts from the last lesson on the cards (1-2 words), e.g. '*Class*', '*Method*', '*Inheritance*', '*regular expression*'. Each student gets a card at random. An additional card gets attached to the board or laid on the ground. Anyone who can explain a connection from his card to an already attached one may place his card in a domino-like manner. Place the first card and explanation yourself. This process continues, until everybody has placed his card. Students may cooperate on difficult topics.

Alternatively (and a nice set-up for an entire lesson) the class can sit in a circle with the empty space in the middle for the Domino.

**Time:** 15'

### ***Recipe 6.2: Crossword***

**Material:**

board

**Method:**

Similar to Domino, but requires less preparation. A single word is written in capital letters in the middle of the board. The students are asked one by one to name and explain a piece of content from the last lesson. They also should indicate, where the name (one word) should be written to the board in a cross-word manner. It is not necessary to take care whether 'random words' emerging between adjacent letters in the cross-word make any sense.

Because the result of this is a quite impressively filled board, this recipe is very useful to motivate students who feel insecure.

**Time:** 15'

### ***Recipe 6.3: Return Home***

**Material:**

None

**Method:**

Ask students how they will explain to their parents the contents of today's seminar.

**Time:** 10'

### ***Recipe 6.4: Pro and Contra***

**Material:**

board

**Method:**

Write a statement on the board that could be interpreted in more than one way, like "Powerpoint makes presentations worse.", "You can write very fast programs in Python", or "viruses are alive". Divide the class into two groups. One collects arguments supporting the statements, the other against it. Have them read their arguments and give reasons. If you like, you may let the group discuss freely.

**Time:** 15'

### ***Recipe 6.5: A Fistful of Explanations***

**Material:**

board

**Method:**

Write a number of concepts on the board (number of students+1). Have everybody explain one of them within two minutes.

**Time:** 2' per student.

## 7. Enforce good Student Presentations

Bad presentations by students (and lecturers) cost lots of time and nerves. Main reasons for a bad presentation are bad preparation, bad presenting or excessive length. Frequently, time is invested in making handouts on cost of the presentation quality. These handouts often are a support for the presenter rather than for his audience. Much of these problems can be avoided by stating clearly at the very beginning of the course, how a presentation has to look like, and to enforce this strictly.

### **Recipe 7.1: Rules for presentations**

**Material:**

board  
stop-watch

**Method:**

Announce at the beginning of the course what you expect from presenters. In practice, always decide in favor of the student, but don't reveal this too early. Write the rules on the board that they can be read clearly by everyone.

**Rules:**

1. A presentation ends after eight minutes.  
(During presentations, set your stop-watch to ten minutes, but don't tell anyone. Warn one minute before the end.)
2. A presentation consists of exactly one diagram on the board, that is to be put there **before** the presentation.
3. The text is to be spoken freely, not read from paper.
4. There will be no handouts.
5. At the end, a three-sentence summary will be given.
6. There will be short, constructive criticism. (Means, you mention something that was good, and where the student can improve himself).

Because of the diagrams, presentations should be at the beginning of a lesson or after the break.

**Time:** 15' per presentation

## 8. Get Feedback

It is crucial that your students feel that you are taking them seriously - I assume you do if you are still reading. The best way to do this is to ask for their opinion. You can be sure they have one, but they might be shy to express it. You should get feedback at least three times in a course or lecture series: The first time as soon as the students know what the course is about. The second time in the middle to check whether you are on the right track. The third time at the end to collect useful suggestions for the next course (and compliments).

### **Recipe 8.1: Closed poll**

**Material:**

small pieces of paper.  
hat or small box.

**Method:**

Write an open question on the board regarding the course, like "*What do you want to learn tomorrow?*", "*What was the most difficult thing today?*". Share pieces of paper among the students. Let them write whatever they want and collect the papers in the box. Leave them in the box **until all students are out of sight**. This method works even

for very shy students. Don't ask *"How do you like the course so far?"*, unless you are really in desperate need to boost your ego.

**Time:** 5'

### **Recipe 8.2: Only one cross each.**

**Material:**

board  
crayon, marker or magnets.

**Method:**

Write a question on top of the board and an empty scatterplot below. The axes are labeled with two scalar fields you are interested in, for instance: X- *"How much did you understand so far?"*/Y- *"Would you like more examples – exercises"*. Each student makes exactly one cross/places a magnet.

**Variation:**

To let the students make a choice among 5-7 'most wanted' topics, you can use a bar or pie-chart-like setup.

**Time:** 5'

### **Recipe 8.3: Burning for Python**

**Material:**

box of matches  
cup with water

**Method:**

Ask *"What will you remember about the course?"*. Everybody may say whatever he wants, but only as long as he is holding a burning match. There is one match per person. This is best performed sitting in a circle, the person left to the docent starting, and the docent coming last. The cup should be held close in order to dispose the match quickly.

**Time:** 10'

**Hint:** A trick only to reveal to another lecturer is that you can control the time you speak by the direction you hold the match.

## **Appendix**

### **1. Exercises on String handling**

#### **Exercise 1.1: String handling using standard functions I**

Read one of the first eight sections in the Python Phrasebook, chapter 2. Understand the code, test it on the command line and explain it by writing up to three example lines of Python code on the board.

#### **Exercise 1.2: String handling using standard functions II**

Read one of the recipes 3.1-3.10 in the Python Cookbook, chapter 3. Understand the code, test it on the command line and explain it to the class writing a few lines of Python code on the board.

### **Exercise 1.3: Printing strings**

Write a function `print_item(name,value,size)`, that creates a nicely formatted one-line string for a list containing the following attributes of an item: the name of the item [string], its value [float] in euros and zlotys (exchange rate is 3.65zł:1€), and its size [integer from 1-5]. Create a simple graphical representation of the size in the string, e.g. one '#' symbol for each size.

### **Exercise 1.4: Regular expressions I**

Write a program that reads the `thief_text.txt` file. Use the regular expression `re.search()` functions to identify all of the lines with size information about items at the end of the file. Print the matching lines, but take care not to detect the false positives (at the bottom of the file).

### **Exercise 1.5: Regular expressions II**

Write a program that reads the `thief_text.txt` file. Use the regular expression `findall()` function to find all of the item-value tuples given in the text (e.g. "cash (12000zł)").

### **Exercise 1.6: Regular expressions III**

Write a program that recognizes the item-value tuples in the `thief_text.txt` file (like in 1.5). Include round brackets in your regular expression, so that the results can be dissected into name and value. Convert the value to a float and return (name, value) tuples.

## **2. Exercises on Classes**

### **Exercise 2.1: An Item class**

Write a class 'Item' that contains one of the items for the backpack problem. An item has at least three attributes: a name, a size and a value. These attributes should be supplied to the constructor as parameters. Implement a `__str__()` method to return a string representation, and a `get_value()` method that can handle two different currencies.

### **Exercise 2.2 An ItemList class**

Write a class `ItemList` that contains zero to many `Item` objects. Implement an `add_item()` method, and a `__len__()` method that returns the number of added items. Make the class printable, like in Exercise 2.1 so that it is printed in a nice way. What data structure would fit these needs best?

### **Exercise 2.3 A Backpack class**

Write a class `Backpack` that can be used to store a limited number of items. The `Backpack` should be a subclass of `ItemList`. The size of a `Backpack` should be limited, and an item should only be added if the total sum of item sizes does not exceed the backpack size.

Which methods or attributes from the `ItemList` class are useful? Which new attributes or methods should the backpack have? Implement a `get_value()` method that returns the total value of the backpack's contents. Also implement a `does_item_fit()` method that checks whether there is sufficient space for a given item inside the backpack.

#### **Hints:**

- What should happen if the backpack gets too full. Consider defining and throwing a special Exception class for that.

- What happens if the thief tries to insert the same Item twice into his backpack?

### 3. Exercises on Reading and Writing Files

#### ***Exercise 3.1: Creating random data***

Write a function that creates a list of Item objects having random value and size. The value should be a float in the range of 10-5000, and size should be an integer in the range of 1-5. Use the Item class and the random module. Test the procedure by generating lists of 10, 100 and 1000 items. Write all items to a text file that contains one item in each line with name, size and value separated by tabulators (\t).

#### ***Exercise 3.2: Creating non-random data***

Write a function that creates up to 100 Item objects. The function should take the number of Items to generate as a parameter. It is up to you, what rules you use to generate name, value and size of items, using the same limitations as in 3.1, as long as you don't write 100 items explicitly. Write all items to a text file that contains one item in each line with name, size and value separated by tabulators (\t).

#### ***Exercise 3.3: Read a tab-separated file***

Write a function that reads one of the text files from 3.1 or 3.2. It should parse all the lines, cut them using the string.split() function. Create Item objects from the data, and put them into an ItemList object. Return the ItemList object.

### 4. Exercises on Data Structures and OOP

The exercises are meant to be split up among the class. The exact distribution of groups and exercises depends on the composition and abilities of the class.

#### ***Exercise 4.1: Write a stack class***

##### **Task:**

Implement a Python class „Stack“ that can perform stack operations. Implement the push(value) and pop() methods, and a constructor \_\_init\_\_() that creates an empty stack. If the stack is empty, pop() should create an Exception. Write code that tests whether the class works correctly. Create a poster that explains how the Stack class is used and how it works.

##### **Material:**

- Python Tutorial, chapters on classes and exceptions.
- big sheet of paper, marker pen.

##### **Hints:**

- Before you start, define an internal representation of the data. It is up to you what data types you use in the class. The course will probably want to know which representation you decided for. You are encouraged to discuss this topic with the people implementing queue and tree classes.
- Is it important for your implementation, what kind of data the user feeds by push() and

pop()?

- Don't forget 'self' as the first parameter in each method of a class.

### **Exercise 4.2: Write a queue class using iterators**

#### **Task:**

Implement a Python class „Queue“ that can perform queue operations. Implement the pop() method. If the queue is empty, pop() should create an Exception. To use the queue in a loop, as in „for value in queue“, make the class iterable. This means implementing the \_\_iter\_\_() and next() methods. Write code that tests whether the queue works correctly. Create a poster that explains what an iterator is and how to use it.

#### **Material:**

- Python Tutorial, chapters on Iterators and Exceptions.
- big sheet of paper
- marker pen.

#### **Hints:**

- After the full for loop, the queue should be empty.
- What happens if the queue runs empty? Can you iterate over an empty queue?
- Is it important for your implementation, what kind of data the user feeds by push() and pop()?
- What happens if someone pushes a None object to the queue?

### **Exercise 4.3: Write a general order tree class**

#### **Task:**

Implement a Python class „GeneralOrderedTree“ for a tree node that can perform tree operations. Implement the methods get\_children(), add\_child(), remove\_child() get\_value() and set\_value() methods. Write code that creates the tree from the slides. Create a poster that explains how the Tree class is used and what recursion is.

#### **Material:**

- big sheet of paper, marker pen.

#### **Hints:**

- A tree can be defined recursively. This means you need to implement the node only once, because all tree nodes behave basically the same:

*„A tree is a root node that contains zero to many trees.“*

- Why is it a **very bad idea** to add a tree as it's own child?

### **Exercise 4.4: Write a binary tree class**

#### **Task:**

Implement a Python class „BinaryTree“ that can perform tree operations. Use a dictionary to store

the nodes of the tree. Use integer indices to access the entire data, as described in the supplementary documentation. Implement the `add_left_child()`, `add_right_child()`, `get_left_child()`, `get_right_child()`, `get_value()` and `set_value()` methods. Write code that demonstrates whether the tree class works correctly. Create a poster that explains the operations that you made on the dictionary.

**Material:**

- supplementary material on representing binary trees.
- big sheet of paper, marker pen.

**Hints:**

- How can you remove children from the tree?
- In what order do the nodes appear if you call the `keys()` method of the dictionary?

***Exercise 4.5: Traversing a tree***

**Task:**

Add a `traverse()` method to the `BinaryTree` class that visits all nodes, and collects their values into a queue. It should return the queue. Your program should print the contents of the queue to make sure no node was forgotten. Find out which traversal method you actually wrote using the supplementary material. Give a three-minute oral summary of your traversal method and your implementation.

**Material:**

- supplementary material on tree traversal methods.
- Python Tutorial on classes and inheritance.

**Hints:**

- Use a recursive method for traversal.
- If you use a stack instead of the queue, how does the order of the elements change? Which traversal method is it then?

***Exercise 4.6: Searching in a tree***

**Task:**

Add a `search(value)` method to the `BinaryTree` class. This method looks for the value in all nodes, and returns the node where it was found, or `None`. Test your method and compare it to the tree traversal methods in the supplementary material. Give a three-minute oral summary of your program.

**Material:**

- supplementary material on tree traversal methods.
- Python Tutorial on classes and inheritance.

**Hints:**

- Use a recursive method for searching.
- If you use a stack instead of the queue, how does the order of the elements change? Which traversal method is it then?

### **Easy Exercise 4.7: Special Methods**

#### **Task:**

Implement the following methods for one of the Stack, Queue or Tree classes:

- `__str__()` - returns a string. Makes any instance `x` of the class usable by `print x`.
- `__len__()` - returns an integer containing the number of items. Makes any instance `x` of the class usable by `len(x)`.
- `__getitem__(name)` – returns an item with the given name from the data structure. Makes any instance `x` of the class usable by `x[name]`

### **Advanced Exercise 4.8: Sort a binary tree**

#### **Task:**

Complete the implementation of the Tree Selection Sort algorithm described in supplementary material. Make it working for a Binary Tree. Collect the sorted items into a Queue object, and print the contents of the queue. Be able to explain the most important sections of your source code briefly.

#### **Material:**

- BinaryTree class from one of the previous exercises.
- supplementary material on Tree Selection Sort.

#### **Hints:**

- Remember that the binary tree is represented by a dictionary. This might make filling the tree with random numbers easier.
- Can you extend the procedure to any number of leaf nodes?
- Compare the performance of your sorting algorithm to the standard Python `list.sort()`.

### **Advanced Exercise 4.9: An abstract superclass**

#### **Task:**

An abstract class is a class that can do nothing on its own, but can be used to create useful filial classes with similar properties. The Queue and Stack classes have some similar properties. Compare them, and create an abstract superclass for them that contains everything that these classes have in common. Give a three-minute oral summary of your method.

#### **Material:**

- Python tutorial on classes and inheritance.

#### **Hints:**

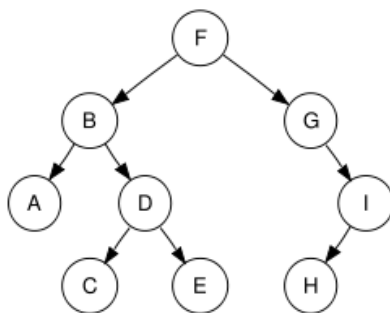
- In some cases, it is important that an abstract class makes sure that a certain method (e.g. `pop()`) exists, not that it does something. Then, it is useful to define a method body in the abstract class that consists of a „pass“ command only.

## Supplementary Material: Tree Traversal

*Wikipedia:* Tree traversal refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well. Compared to linear data structures like linked lists and one dimensional arrays, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed with the order that they are performed defining the traversal type. These steps are: Performing an action on the current node (referred to as "visiting" the node); or repeating the process with the subtrees rooted at our left and right children. Thus the process is most easily described through recursion.

- To traverse a non-empty binary tree in **preorder**, we perform the following three operations: 1. Visit the root. 2. Traverse the left subtree in preorder. 3. Traverse the right subtree in preorder.
- To traverse a non-empty binary tree in **inorder**, perform the following operations: 1. Traverse the left subtree in inorder. 2. Visit the root. 3. Traverse the right subtree in inorder.
- To traverse a non-empty binary tree in **postorder**, perform the following operations: 1. Traverse the left subtree in postorder. 2. Traverse the right subtree in postorder. 3. Visit the root.
- Finally, trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level.

*Tree traversal*



In this [binary search tree](#),

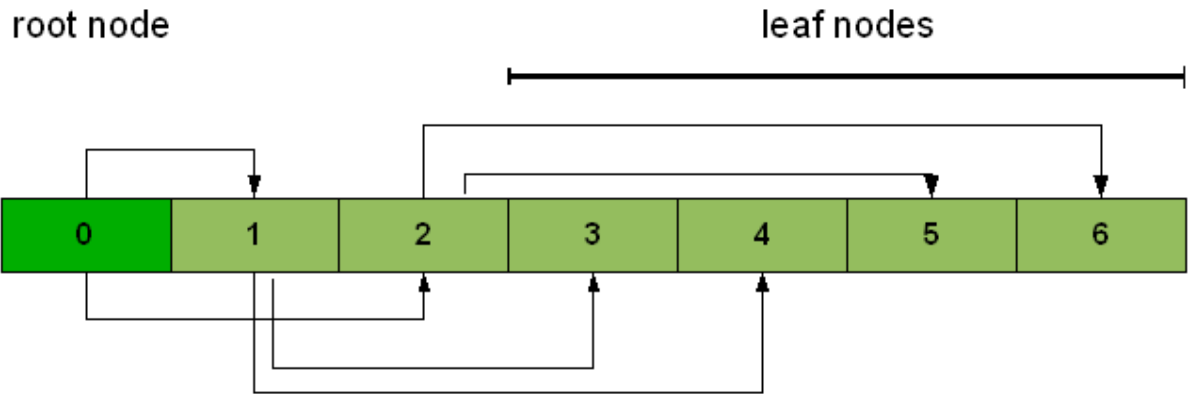
V = visit, L = left, R = right

- Preorder (VLR) traversal yields: F, B, A, D, C, E, G, I, H
- In-order (LVR) traversal yields: A, B, C, D, E, F, G, H, I
  - Note that the in-order traversal of a binary search tree yields an ordered list
- Postorder (LRV) traversal yields: A, C, E, D, B, H, I, G, F
- Level-order traversal yields: F, B, G, A, D, I, C, E, H

## Supplementary Material: Representing a Binary Tree as an Array

*Wikipedia:* Binary trees can also be stored as an implicit data structure in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index  $i$ , its children are found at indices  $2i+1$  and  $2i+2$ , while its parent (if any) is found at index  $(i-1)//2$  (assuming the root has index zero). This method benefits from more compact storage. However, it is expensive to grow and wastes space proportional to  $2h - n$  for a tree of height  $h$  with  $n$  nodes.

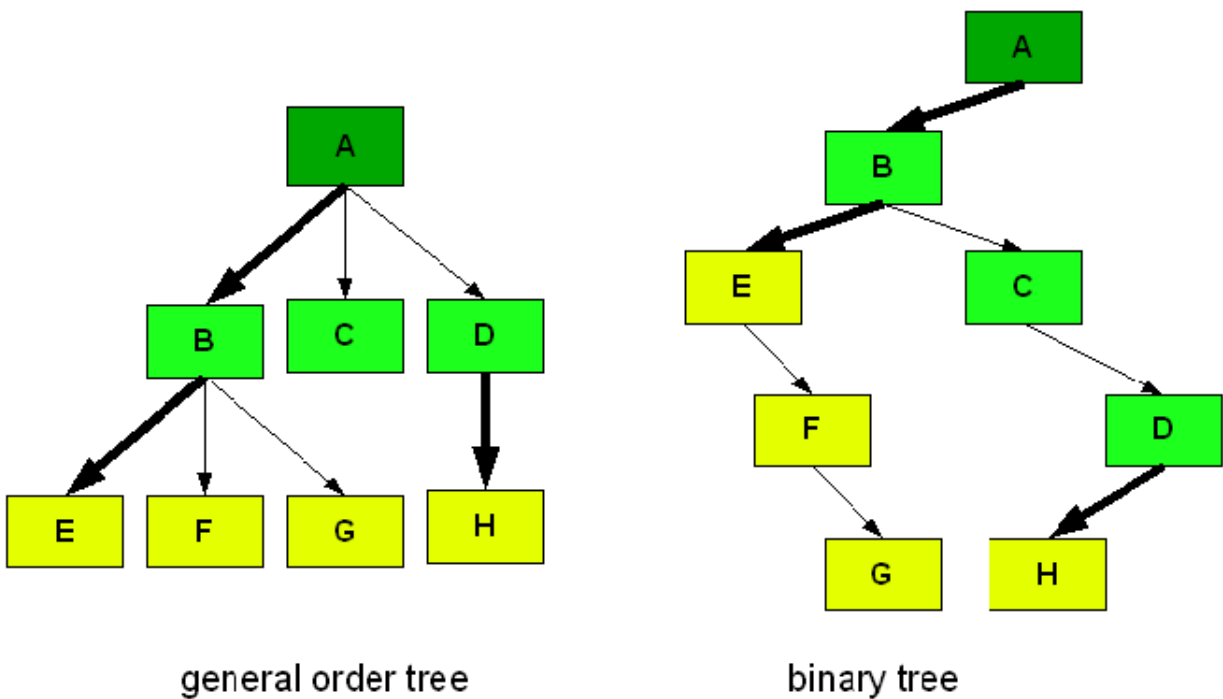
*Representation of a binary tree as an array*



**Supplementary Material: Representing a General Ordered Tree by a Binary Tree**

*Wikipedia:* There is a one-to-one mapping between general ordered trees and binary trees. Each node  $N$  in the ordered tree corresponds to a node  $N'$  in the binary tree; the left child of  $N'$  is the node corresponding to the first child of  $N$ , and the right child of  $N'$  is the node corresponding to  $N$ 's next sibling - that is, the next node in order among the children of the parent of  $N$ .

For example, in the figure, the ordered tree on the left can be converted into the binary tree on the right. The binary tree can be thought of as the original tree tilted sideways, with the thick edges representing the first childs and the thin edges representing the next sibling.



### Supplementary Material: Tree Selection Sort

There are many different algorithms for sorting things, like Selection Sort, Bubble Sort, Quicksort, Heapsort, Mergesort etc. They significantly differ in performance. The two most important measures of a sorting algorithm are the number of comparisons and move operations between items with respect to the number of items. Factors that play a role in choosing the optimal algorithm are the type of data, whether it is partially sorted, and of course the length of the list to sort. In Python, all lists have by default a Quicksort function implemented as the `list.sort()` function.

Sorting data in a binary tree representation is often more efficient than operating on a plain list. An easy possibility is the Tree Selection Sort algorithm, whose pseudocode is given below. It assumes that the data is stored in the leafs of a binary tree, as one value for each node. All nodes that are no leafs contain empty values:

1. Create an empty queue that collects the sorted values.
2. Create a variable for the current node, that contains the node currently processed.
3. If the root node contains a value, push it to the queue. Set the root nodes' value to empty.
4. Start with the root node as the current node.
5. If the current node has no children, delete that node. Remove that node from the parent. Go back to 3.
6. If the current node has one child, set the current nodes' value to that of the child (empty or not). Set the child's value to empty. Set the current node to the child. Go back to 5.
7. If the current node has a child node with empty value, set the current node to that child. Go back to 5. (*if this applies to both childs, any of them will do*).
8. There are two child nodes, and both of them are not empty. Compare their values. Set the current nodes' value to that of the smaller child. Set that child's value to empty. Set the current node to that child. Go back to 5.
9. As long as the root node has children, continue with 3.-8.

## 5) Exercises on Dynamic Programming

### Summary: The Backpack Problem

A thief has broken into a house. He finds a number of items that would be worth taking with him. Since the thief is a tough professional, he can estimate the market value of each item precisely. Unfortunately, he will have to croug through a sewer tube on his way out. This limits the total volume of stuff he can carry with him. So, he is thinking, what things to put into his backpack,

#### Basic item list

item	size	value	item	size	value
TV	3	200€	silver spoons	1	400€
stereo	2	1000€	jewellery	2	2000€
antic vase	3	1500€	camera	1	100€

DVD	2	600€	cash	1	3000€
-----	---	------	------	---	-------

### ***Pseudocode of the dynamic programming algorithm***

- Create a list that will contain the best combination of items for a given backpack size
- Insert an empty combination for backpack size 0.
- Start with a backpack of size 1.
- Set the best combination for current size-1 as the candidate for the best combination for the current size.
- Go through the items
- Create a combination of this item plus the best combination from size (current size size of the current item).
- If this combination is more valuable than the candidate combination, replace the candidate combination by this one.
- Increase the current backpack size by 1
- start five steps above again.
- Return the best combination for the desired backpack size.

### ***Advanced Exercise 5.1: Dynamic Programming***

Write a program that solves the backpack problem. A draft implementation of the algorithm exists already in `BackpackProblem.py`. Make sure you use the `Item`, `ItemList` and `Backpack` classes. Find out whether you need to add any methods to make the procedure working. Test it on a small set of items.

#### **Important Steps:**

- Familiarize yourself with the `Item` and `Backpack` classes first.
- Go through the pseudocode and make sure you understood what is meant by each line.
- Don't forget to let the `__main__` part of the program feed some reasonable test data.

### ***Advanced Exercise 5.2: Probing the algorithm***

Run the backpack algorithm with different sets of data generated by exercises 4.1 and 4.2. Try different small and large sets. Measure the time the programs need to run, and write them up in a table. The Unix command line tool 'time' is very useful for that. How does the time relate to the number of items?

Use the Python profiler to find out where the program spends most time. Identify bottlenecks. Discuss the results and potential solutions.

## **Glossary**

*array*

computer science term for a list.

*index*

the integer indicating the position in a list.

<i>root</i>	first node of a tree.
<i>child node</i>	connected node one level down the tree.
<i>sibling nodes</i>	nodes that have the same parent.
<i>leaf</i>	any node without further child nodes attached.
<i>height</i>	the number of nodes from the furthest leaf to the root.
<i>binary tree</i>	tree where a node is connected to two child nodes at most.
<i>general order tree</i>	tree where a node can have any number of child nodes.
<i>complete binary tree</i>	tree where all leafs have the same distance to the root.