IWGS – Informatische Werkzeuge in den Geistes- und Sozialwissenschaften – WS 2018/19

Michael Kohlhase
Informatik, FAU Erlangen-Nürnberg

February 10, 2019

Contents

Assignment 1 (First Steps with Python) – Given 26. 10. 2018, Due 9. 11. 2018

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 1.1 (Hello World)

Go to `http://pythonanywhere.com`, make an beginners account there and set the teacher's account to iwgsTeacher. 20pt

Write an extended "Hello World Program" in a file called exthello.py. The program should print information about you and your account. Specifically, the information should be:

```
Hello World!  I am <your name>.
My account name is <account>, and my teacher is iwgsTeacher.
```

As you have given the teacher account access to our account, we can (and will, for grading) check your program on your account.

Submit only your account name to StudOn for this exercise.

Solution: This is almost the same as in the slides.

```
print("Hello, World! I am Example Student. My account is exampleStudent,
and my teacher is iwgsTeacher.")
```

Problem 1.2 (Variable Assignment and Output)

Write a program in python that calculates the total number of seconds in a leap year, stores the result in a variable and then displays that to the user. 20pt

Solution: We assign the (result of the) computation (60∗60∗24∗366) (seconds in a minute, minutes in an hour, hours in a day, days in a leap year) to a variable called seconds and then use that as an argument to the **print** operator.

```
seconds = 60∗60∗24∗366
print(seconds)
```

Problem 1.3 (Variable Reuse)

Programming often has a efficiency as one of its goals. After all, why go through the trouble of telling a computer how to do something, if you could do it better and quicker yourself? 20pt

Write a program in python that prints the string "supercalifragilisticexpialidocious" five times, but *without* typing the word five times yourself.

Solution: We first assign the string supercalifragilisticexpialidocious to a variable text and then use that as an argument to the **print** operator instead of the string itself.

```
text = "supercalifragilisticexpialidocious"
print(text)
print(text)
print(text)
print(text)
print(text)
```

Eine alternative Lösung ist auch:

```
text = "supercalifragilisticexpialidocious"
print(text * 5)
```

Problem 1.4 (Human Readable Time)

In programming, it is often the case that your program collects a lot of data from various sources. It then becomes essential to present this data in a way that the user (usually a human!) can easily understand. For example, most humans don't know how long a longer timespan is if it is given only in seconds. 20pt

Write a program in python that first initialises a variable seconds = 1234567. Then, the program should calculate and print how long this timespan is in days, hours, minutes and seconds instead of just seconds.

Solution: We begin by assigning the variable seconds as instructed.

Then, we use *integer division* to calculate how many days fit into the timespan by deviding the total number of seconds by the amount of seconds in a day. Then we substract that amount from the amount of seconds, leaving us with a time interval of less than one day. We do the same thing for hours and minutes.

Lastly, we use the **print** operator to display our results. The only change to previous exercises being that we give it multiple arguments instead of just one.

```
seconds = 1234567

days = seconds // (60*60*24)
seconds -= days * (60*60*24)
hours = seconds // (60*60)
seconds -= hours * (60*60)
minutes = seconds // 60
seconds -= minutes * 60
print("Saved time:",days,"days,",hours,"hours",minutes,"minutes",seconds,"seconds")
```

Problem 1.5 (String Presentation)

Keeping with the importance of well-presented information: You can use certain special symbols in strings to give them a better formatting when they are ultimately printed. For example, when you put "\n" into a string, instead of printing these symbols, the output switches to a *new line*. 20pt

Write a python program that prints your favourite haiku (a poem with five syllables on the first line, seven on the second and five on the third) on three three lines, but using only *one* **print** statement.

P.S.: If you don't have a favourite haiku and can't think of one yourself, you can use this one:

My cow gives less milk,
now that it has been eaten,
by a fierce dragon.

Solution: This is just printing a string like before, but with "\n" inserted into the string at the line breaks.

```
print("My cow gives less milk,\nnow that it has been eaten,\nby a fierce dragon.")
```

"If you are learning programming (or anything else) what you do when you get "stuck" will be the sole determinant of your success."
Sarah Mount

Assignment 2 (Control Flow in Python) – Given 09.11.2018, Due 16.11.2018

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 2.1 (User Input)

One of the most important things to learn about a programming language is how to get input from the user in front of the screen. In python, one way of doing this is the input instruction. 15pt

For example: if you write answer = input("Do you like sharks?"), this will print the message you gave ("Do you like sharks?"), wait for the user to submit a response and store it as a string in the variable answer when you run the program. You can then use it like any other value stored in a variable.

Write a simple program that prints a generic greeting message, then asks the user to input their name, stores the input in a variable and then finishes with a goodbye message that uses the name the user gave.

Solution: After printing the greeting message with a **print** statement, we use an input statement with a fitting message asking for the name. We want to use the answer the user gave later, so we bind this input to a variable (called name in this case). Lastly, we print a goodbye message. If we give **print** two arguments, the message itself (as a string) and the variable we used before, it will print both together.

```
print("Hi there!")
name = input("What's your name? ")
print("Bye, ", name)
```

Problem 2.2 (Simple Branching)

The next important concept is control flow. A program that always does the same thing gets boring fast. We want to write programs that do different things under different circumstances. In python, one way to do this are conditional statements. 25pt

Write a python program that asks the user if they have a pet. If their answer was "yes", the program should ask what kind of pet they have. Since sloths are the cutest animals (at least for this exercise), the program should print "awww!" if the user's second answer was "sloth" and "cool!" if it was something else. If the user does not answer with "yes" the first time around, the program should quit with a goodbye message.

Solution: This exercise also relies heavily in input statements. We again save the results in a variable. We then use boolean expressions (expressions that evaluate to True or False) that compare these to the expected answers of "yes" and "sloth" with the == operator for the two nested **if**-statements.

When using **if**s in python, do not forget the ":" after the boolean expression and to *indent* the code blocks that are to be executed only if the expression evaluates to True. Otherwise, you will produce an error.

```
ownsPet = input("Do you own a pet? ")
if ownsPet == "yes" :
    pet = input("What kind of pet? ")
    if pet == "sloth" :
        print("awww!")
    else :
        print("cool!")
else :
    print("Goodbye!")
```

Problem 2.3 (Simple Looping)

Computers are very good at doing the same thing over and over again without complaining or messing up. Humans are not. In python, we can use a loops if we want something done multiple times. 25pt

Suppose your boss wants the string "Programming is cool!" printed exactly 1337 times (for some reason...). Typing up the string yourself takes about nine seconds each time, printing it in a loop takes no time.

To save time, write a python program that prints the sentence "Programming is cool!" 1337 times using a loop. Your program should also keep track of (store in a variable) how much time the loop saved the programmer in total (9 seconds per iteration of the loop). Print this value after the loop finishes.

Solution: First, we need to create a variable in which we will keep track of our saved time. It starts out as 0.

Then, we use a loop as instructed. If we know in advance, how often we want something done, like in this example, a for loop is most fitting. In python, the way to go about this, is using a range. If we want something done n times, the syntax is `for x in range(n):` (where x is a variable, so it could also have any other name). In the (indented!) code block in the loop (called the *body*), you can then use the variable like any other and it will have the value 0 in the first iteration, 1 in the second, and so on...

In every iteration, we want to print the target string once and increase our time-saving variable by nine because we saved nine seconds. In this example solution, we use the += notation for that. `secondsSaved += 9` is just another (shorter) way to write `secondsSaved = secondsSaved + 9`.

Lastly, we print the amount of saved seconds alltogether.

```
secondsSaved = 0
for x in range(1337) :
    print("Programming is cool!")
    secondsSaved += 9

print("seconds saved:", str(secondsSaved))
```

Problem 2.4 (Temperature Conversion)

Write two python programs, named celsius2fahrenheit and fahrenheit2celsius, that given a number as input from the user convert it to the respective other temperature scale and print the result. 35pt

The conversion formulas are as follows:

$$[^\circ C] = ([^\circ F] - 32) \cdot \frac{5}{9} \qquad [^\circ F] = [^\circ C] \cdot \frac{9}{5} + 32$$

Remember that input will save the input as text, not as a number. You can convert a string to a number using the function float.

Example: float("3.1415") will evaluate to the *number* 3.1415. If the text given to float does not actually represent a number (e.g. float("bad")), python will throw an error.

Afterwards, please test your programs against another converter (easily found via your internet search engine of choice) to make sure that your functions produce the correct results.

Solution: This exercie also uses input to grab some input from the user, with the key difference being that we convert it from the pure text to an actual number via the function float afterwards. This is an important step that you cannot leave out. Numbers and text are two very different things in a computer and if you try to do arithmetic with text, it will not work, even if the text is a textual representation of a number.

Once that is done, the rest is simple arithmetic following the formulas and another simple print statement like already discussed above.

```
#celsius2fahrenheit
celsius = float(input("Please input temperature in degrees Celsius: "))
fahrenheit = celsius * 9 / 5 + 32
print("That's",fahrenheit,"degrees Fahrenheit.")

#fahrenheit2celsius
fahrenheit = float(input("Please input temperature in degrees Fahrenheit: "))
celsius = (fahrenheit - 32) * 5 / 9
print("That's",celsius,"degrees Celsius.")
```

Assignment 3 (Data Structures) – Given 16.11.2018, Due 23.11.2018

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 3.1 (Basic Lists)

When working with lists, the first and the last elements of the list are often of special interest or significance. 20pt

1. Write a python function that, when given a list as a parameter, prints (on two separate lines, with some explanatory text) the first and last elements of the list.
2. Is it possible to do this without looping over the entire list to find the last element?
3. What happens when you give this function a list of only one element?
4. What happens when you give it the empty list?

Solution: In python, you can access certain elements of a list if you write the name of the list, followed by a number in square brackets (e.g. myList[1]). These numbers start with 0 (not 1!) for the first element.

There is also a shorthand for accessing the elements from the end of the list instead of from the start. It's the same as above, but you write a negative number in the square brackets. Here, they start with -1 for the last element (and -2 for the second-to-last and so on). It's not -0 because $-0 = 0$

1. This function will do it:

```
def firstAndLast(lst):
    print("The first element is:", str(lst[0]))
    print("The last element is:", str(lst[-1]))
```

2. Yes, via lst[-1].
3. The first element is also the last one.
4. This will crash the program because there is no "first" element in an empty list.

Problem 3.2 (User Input II)

Often, when you are taking input from the user, it becomes important that the input is one of a certain set of "acceptable" answers. 30pt

Write a python program that asks the user for their favourite deadly sin. If the input it receives is not one of the acceptable answers (i.e. the strings "lust", "gluttony", "greed", "sloth", "wrath", "envy" and "pride"), it should keep asking again and again.

Hint: You can use a loop to achieve this. Using a "sin list" will also be helpful.

When the input is (finally) correct, it should print a message either complimenting or deriding the user on their pick (your choice!).

Solution: The trick here is to use an input *within* a **while**-loop to have the program ask again and again as long as the answer is not acceptable. One way of testing for this is having a list of all acceptable answers and testing with the X **in** Y construct if the given answer is in the list.

For this to work on the first pass, we also have to initialise the answer variable to something (like the empty string in this example) first, because it only gets assigned something in the loop, after it was already checked once.

```
sins = ["lust", "gluttony", "greed", "sloth", "wrath", "envy", "pride"]

answer = ""
while (answer not in sins):
    answer = input("What is your favourite deadly sin? ")

print("Oh! You picked", answer)
print("Excellent choice! But don't let the catholics hear that!")
```

Problem 3.3 (Dictionaries) 30pt

In programming, it is important to gain familiarity with the most commonly used data structures. This exercise will make you more familiar with the dictionary data structure.

1. Write a python dictionary that associates names of famous peoples (i.e. strings as keys) with their year of birth (i.e. ints as values). The entries can be real or fictional people, as long as they have a clear year of birth.

2. Write a program that finds the oldest person (i.e. lowest year of birth) in that dictionary. (How) can you loop over all keys of a dictionary? Finally, your program should print in what year the oldest person in your dictionary was born (it does not have to say who that person is).

Solution: You can write a dictionary in python directly with the name = { key : value, ...} notation.

Afterwards, we need to loop over all keys that the dictionary has. For this we can use the **for** X **in** Y construct, where X is a variable and Y is our dictionary (this also works for lists and some other data structures).

Accessing dictionaries functions similar to accessing lists, but we write keys (which might be of any type) in the square brackets, not just numbers. We can access the dictionary for every key that we loop over and compare to our current best, updating the variable whenever we find a new lowest value.

1. Here is the dictionary

```
dictIdols = { "Yuri Gagarin" : 1934, # First person in space.
              "Neil Armstrong" : 1930, # First person on the moon.
              "Ada Lovelace" : 1815, # Wrote the first computer algorithm.
```

```
                "Alan Turing" : 1912, # Formalised the notion of algorithm.
                "Rosa Parks" : 1913 } # Anti-racism activist.
```

2. and here the program (you should either read the dictionary above via an **import** or put it into one file with the dictionary

```
currentBest = 9999 # one initial value that's higher than all of them.

for person in dictIdols:
    yearOfBirth = dictIdols[person]
    if yearOfBirth < currentBest:
        currentBest = yearOfBirth

print("The oldest person in the database was born in", currentBest)
```

Problem 3.4 (Project 01) 20pt

During the remainder of the semester, you will be developing example applications in teams. Your task for this exercise is to form teams of three and think of an application that you would like to develop. Ideally, this would draw on or somehow be relevant to a background in the humanities.

The application should have to manage some sort of data (text, images, ...), and generate some sort of (simple) interface (web pages, graphical representations, ...).

Hand in

1. a unique name for your project,
2. your team composition, and
3. a short text (maybe two paragraphs) about what kind of application you would want to develop. This should include a description of what kind of user the application is for, how they will interact with it, what they can and cannot do with the program and what kind of data the program will have to manage.

   Example: As our running example, the fictitious IWGS students Nadja, Lucas and Amira will be developing a order management program for the pizza delivery service "Planetary Pizza" (a deliberately not humanities-based topic, so that the example is neutral and different).

   They write down that the user should be able to assemble and send a pizza order, including both pizzas from a menu (loaded from the pizzeria's database) and pizzas that they themselves choose the toppings for. During this process, the user should be able to see their selected pizzas so far and the individual prices, as well as a sum total.

Solution: *No master solution for this problem; there is an example in the problem text above.*

Assignment 4 (Unicode and Number Systems) – Given 23.11.2018, Due 30.11.2018

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 4.1 (Egyptian Hieroglyphs 1: Numerals)

Programming is a versatile discipline and applicable to a lot of very different fields, from space satelites to fast pizza delivery to Egyptian hieroglyphs. In the following exercises, you will take a closer look at the latter to familiarise yourselves with the Unicode encoding. 40pt

The Egyptian numeral system[1] is decimal, like our system, but is not position-based (similar to Roman numerals). Each hieroglyph has a certain Unicode encoding[2], i.e. a certain number that people have agreed upon to represent a certain hieroglyph.

The Egyptian number system is relatively simple (for numbers up to $1,000,000$ or so). Learn about it. Then, write a python function arabic2Egyptian that takes a standard (positive) integer and returns a unicode string of a corresponging Egyptian number.

Note: The code here will be structurally similar to a previous exercise. Also recall that the Universal Character Set assigns every character a hexadecimal number n, e.g. 1F607 (smiling face with halo). If we want to use character n in a string in python, just use "\U0001F607" (i.e. n filled up with leading zeros to make it 8 hex digits).

Note: Note that we will *not* be awarding / deducting points on precise hieroglyph choice. As long as the hieroglyphs you chose roughly align with those presented in the number systems article, we will assume them correct. This goes for all exercises on this sheet.

Solution: The code for this exercise works similar to that of the human readable time exercise. When you want to say "300" in egyptian numerals, you take the numeral for "100" three times and so on. So you can just do the following for every numeral: add the corresponding glyph to an (initially empty) string as often as the numeric value fits in your number (found out by integer division). After that, reduce your number by the appropriate amount (like taking the modulo of your current number and the numeric value of the glyph).

[1] See, for example: `https://en.wikipedia.org/wiki/Egyptian_numerals`

[2] See `https://en.wikipedia.org/wiki/Egyptian_Hieroglyphs_(Unicode_block)` for details

The short and sweet of using unicode glyphs in python is that you can do so by writing "\u0000" for a 16-bit-Unicode symbol, where the 0s are replaced by the correct numbers. For example, "\u0394" is Δ.

For the egyptian hieroglyphs, you will have to use 32-bit-Unicode symbols, which are introduced to python with a "\U" instead of "\u" and take eight numbers instead of four. For example: "\U00013190" is an egyptian tadpole hieroglyph.

You can find more details here: `https://docs.python.org/3/howto/unicode.html`

Taken together, your code might look something like this:

```
def arabic2Egyptian(inp):
  string = ""
  number = inp

  # Heh / Huh, U+13068, 1,000,000
  string += "\U00013068" * (number // 1000000)
  number = number % 1000000

  # Tadpole, U+13190, 100,000
  string += "\U00013190" * (number // 100000)
  number = number % 100000

  # Finger, U+130AD, 10,000
  string += "\U000130AD" * (number // 10000)
  number = number % 10000

  # Lotus Plant, U+131BC, 1,000
  string += "\U000131BC" * (number // 1000)
  number = number % 1000

  # Coil of rope, U+13362, 100
  string += "\U00013362" * (number // 100)
  number = number % 100

  # Hobble, U+13386, 10
  string += "\U00013386" * (number // 10)
  number = number % 10

  # Vertical Stroke, U+133FA, 1
  string += "\U000133FA" * number

  return string
```

Problem 4.2 (Egyptian Hieroglyphs 2: Text)

Supose that word has gotten around that you know how to handle Unicode in python and one of your friends who is also an egyptology enthusiast wants your help. 40pt

The standard method of displaying Egyptian hieroglyphs (etched into stone or clay) can be slow in writing and just remembering longer messages can be hard to do[3]. A digital

[3]Compare "Ente, Auge, Zickzack" (ZDF, German): `https://www.youtube.com/watch?v=SbZXiDE6GO4`

format would be so much simpler!

First, write a python dictionary that associates English or German words (keys) to fitting Unicode symbols (values). Your dictionary obviously does not need to translte *all* hieroglyphs, but should at least inlcude five different ones.

Second, write a program that, using this dictionary, will ask the user again and again for input, looks up the value associated with that input in your dictionary and appends it to a string variable. When some special phrase to end the program is entered (e.g. "exit" or "quit"), the program should print the variable and exit.

This way, you can take a message that's easy to write on a Western keyboard and easily turn it into proper Egyptian hieroglyphs.

Solution: There are two points to this exercise: the dictionary itself and the program to read in user input and "translate" the user input.

You already know from the lecture how to write python dictionaries in principle. Here, you will be matching up strings the user might input with Unicode strings that give the corresponding hieroglyph in the "\U00000000" notation.

One way of writing the program is using a **while**-loop that loops until a certain variable contains the string "exit" (or "quit" or whatever your programmer heart desires). To make sure this does not become an infinite loop, we also assign the user input to that variable in the body of the loop. This way the user can quit the program by entering the string you specified.

The only thing left is to append the corresponding glyph to a string we are using to hold all glyphs (set to the empty string before the loop). For this, we can simply append (e.g. +=) the value we get by looking up the user input in the dictionary we wrote. As a reminder, dictionary lookup works like this: dict[key] and returns the value associated with the used key.

All in all, your program might look like this:

```
dictEn2Eg = { "ente" : "\U00013170",
              "auge" : "\U00013079",
              "zickzack" : "\U00013216",
              "schlange" : "\U00013199",
              "stern" : "\U000131FC",
              "eule" : "\U00013153",
              "spirale" : "\U00013362" } # and so on and so forth...

collection = ""
recentinput = ""

while (recentinput != "exit"):
    recentinput = input("Next glyph: ")
    if (recentinput != "exit"):
        collection += dictEn2Eg[recentinput]

print(collection)
```

Problem 4.3 (Egyptian Hieroglyphs 3: Input Sanitising) 20pt

Whenever you ask a user for input that you want to use in a meaningful way later in your program, it is vital that you make sure the user has actually entered something sensible. Because often, they won't.

Concretely, if you look up a key in a dictionary that was never assigned a value, python will print an error message and your program will crash.

Amend your program from the previous exercise to check if the entered word is actually a key in the dictionary you are using. If it isn't, you can print an error message or simply ask again. Entering garbage should no longer crash your program.

Solution: We use the same program structure like in the last exercise. The only change is, that we're adding one **if**-statement. If we have a dictionary called mydict, the expression x **in** mydict evaluates to True if and only if x is a key of mydict. If the input the user gave is indeed a key, we add the corresponding glyph like before. In all other cases (**else**), we print an error message and restart the loop.

```
# See previous example solution for dictionary.

collection = ""
recentinput = ""

while (recentinput != "exit"):
    recentinput = input("Next glyph: ")
    if (recentinput != "exit"):
        if recentinput in dictEn2Eg:
            collection += dictEn2Eg[recentinput]
        else:
            print("Glyph not available!")

print(collection)
```

* * *

Note: The following is a *bonus* exercise. You can hand in a solution for extra points, but correct solutions to the first three will already count as 100%.

Problem 4.4 (Basechange)

Colours are important for a plethora of things in software development and there are many ways of describing just which colour you are talking about. 40pt

Maybe the most common way to specify a colour is by giving a triple of numbers between 0 and 255, signifying the how strong the red, green and blue (RGB) components in the colour are. Often, these are given as values in base 16 (i.e. 00 to FF).

First, make sure that you understand how a hexadecimal number system works. Then, write a function that takes a string as an argument. This string will only have one (hexadecimal) character, either of the following:

["0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"].

The function should return the *decimal* value of the input as a regular integer.

Then, using the function you just finished, write a program that takes strings of six hexadecimal characters (two for red, two for green and two for blue, in that order, e.g. 00FF88 or 326496) and prints their correct RGB components in decimal.

Solution: The first function is not very complex, but long, since you need to check 16 different possibilities.

The second function accesses the different characters of the string. Every colour (red, green, blue) has two characters, the first of which needs to have its decimal value multiplied by 16 before adding the decial value of the second one (conversion from hexadecimal to decimal) to give the complete value for that color. Do this for all three colours and print the result.

```
def convertOne(inp):
    if inp == "0":
        return 0
    elif inp == "1":
        return 1
    elif inp == "2":
        return 2
    elif inp == "3":
        return 3
    elif inp == "4":
        return 4
    elif inp == "5":
        return 5
    elif inp == "6":
        return 6
    elif inp == "7":
        return 7
    elif inp == "8":
        return 8
    elif inp == "9":
        return 9
    elif inp == "A":
        return 10
    elif inp == "B":
        return 11
    elif inp == "C":
        return 12
    elif inp == "D":
        return 13
    elif inp == "E":
        return 14
    elif inp == "F":
        return 15

def convertAll(inp):
    red = (convertOne(inp[0])*16) + convertOne(inp[1])
    green = (convertOne(inp[2])*16) + convertOne(inp[3])
    blue = (convertOne(inp[4])*16) + convertOne(inp[5])

    print("The red, green, and blue components in decimal are:", red, green, blue)
```

Assignment 5 (Regular Expressions) – Given 01.12.2018, Due 07.12.2018

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 5.1 (Regular Expressions 1) 40pt

In this exercise we will explore regular expressions. Regular expressions allow us to find patterns in a given text and even modify the matched sections. In order to use regular expressions, you need to import the "re" package. This is done by typing "**import** re" at the top of your python file.

In the imperial unit system, weight is measured in pounds (lb). As Central Europeans are more used to expressing weight in kilograms (kg), we will use regular expressions to find occurences of weight measurements in a text and convert it.

Consider the following text[4]:

> Two-thirds of Americans report that their actual weight is more than their ideal weight, although for many, the difference between actual and ideal is only 10 pounds or less. But 30% of women and 18% of men say their current weight is more than 20 pounds more than their ideal weight. The average American today weighs 17 pounds above what he or she considers to be ideal, with women reporting a bigger difference between actual and ideal than men.

Use regular expressions to find all numbers in the text. Use the re.findall() function[5], which returns a list of matches.

Take into consideration, that numbers can consist of more than one digit. Print the list of matches. Amend the program, such that it only matches occurences of pound measurements, i.e. only numbers followed by the string "pounds". The list for the above text should now be ["10 pounds", "20 pounds", "17 pounds"].

In regular expressions, you can group certain parts of the pattern by enclosing it in parentheses. This can be useful, if you want to further process the results of the matching.

Amend your program, such that findall() returns the following list: ["10", "20", "17"]. Note that these are still only the numbers followed by "pounds", but the "pounds"-part is stripped away automatically.

Loop over your list of measurements. For each entry, convert the entry to kilograms using the following formula:

$$[kg] = [lb]/2.2046$$

[4]Source: `https://news.gallup.com/poll/102919/average-american-weighs-pounds-more-than-ideal.aspx`

[5]`https://docs.python.org/3/library/re.html#re.findall`

Print the conversion with some explanatory text, i.e. "10lb are 4.535970244035199kg".

Solution: This is mostly basic regex work and should not be too difficult to figure out after reading the documentation and playing around with regular expressions a bit.

The only part that is a little tricky is that using parentheses around "\d+" (which, by itself, captures one or more digits) creates a "capture group". This has the effect, that we can add pounds after the capture group and while now this regex will *match* only full weight measurements (but not, say the percentages in the text), only the numbers (and not the full string including pounds) will be returned as result. Which is what we want.

```
# Don't forget to import the library.
import re

# Save string to be searched.
haystack = """Two-thirds of Americans report that their actual weight is
 more than their ideal weight, although for many, the difference between
 actual and ideal is only 10 pounds or less. But 30% of women and 18% of
 men say their current weight is more than 20 pounds more than their
 ideal weight. The average American today weighs 17 pounds above what
 he or she considers to be ideal, with women reporting a bigger
 difference between actual and ideal than men."""

# Use findall from re to find all pound measurements.
#
# "\d" matches any digit, "\d+" matches one or more digits,
# "\d+ pounds" matches the weigth measurements in pounds, but not percentages.
# "(\d+) pounds" does the same, but only returns the numbers.
pound_values = re.findall(r"(\d+) pounds", haystack)

for lbs in pound_values: # Loop over all results in pound_values-
    kg = float(lbs) / 2.2046 # Convert to kg.
    print(lbs, "pounds are", kg, "kg") # Display helpful text.
```

Problem 5.2 (Regular Expressions 2) 60pt

In the real world, data processed by computers often comes from files read from the hard disk. Consider the following spreadsheet table:

	A	B	C
1	Dentist	11/29/2018	Example Str. 22
2	Exam	2/7/2019	Kollegienhaus
3	Hair cut	12/3/2018	Example Str. 25

It lists appointments line by line. Each line consists of the type of appointment, the date and the place. A common data format is the `CSV` file format. Most spreadsheets (like LibreOffice Calc or Microsoft Excel) support exporting to this format.

The resulting `CSV` file (also supplied for this exercise) looks like this:

```
Dentist;11/29/2018;Example Str. 22
```

```
Exam;7/2/2019;Kollegienhaus
Hair cut;12/3/2018;Example Str. 25
```

CSV is short for "Comma Separated Values". As the name implies it lists the entries, separated by commas (actually it's semicolons in this case).

The dates in this example are given in the American notation: Month/Day/Year. We will use regular expressions to convert it into German notation: Day.Month.Year, i.e. day before month and separated by dots instead of slashes.

Open the file using python's File I/O (input/output) functionality[6]. Read the whole file using the readlines() function, which returns a list of lines. Print this list.

Now loop over the list and perform the following for each entry: Use the string split() method[7] to separate individual entries at the semicolons.

For example, splitting the entry "Dentist;11/29/2018;Example Str. 22" at the semicolons should give you the list ["Dentist", "11/29/2018", "Example Str. 22"].

The second value is the date we would like to convert. Use the re.sub() function[8] to extract the day, month and year and reassemble them in the German notation. Afterward print some useful text for the appointment containing the converted date.

Solution: This exercise is a little trickier than the one before. We'll focus on the part about regular expressions here.

In the last exercise, we introduced "capture groups", that can match on a value in a certain context (like being followed by the string "pounds") and hang on only to the value, without the context.

It turns out we can also give these capture groups names (that we can then refer back to later, when we need them) by writing (?P<name>...) where name is the name we want to use and ... is the actual pattern we want to match against.

Also, we need to know that p{m,n} means "at least m and at most n repetitions of p. p{k} means exactly k repetitions. At least that is the way the example solution below does it. There are other ways. For example, you could also write \d\d\d\d instead of \d{4} and so on.

All of this and more is also listed in the documentation and on the website that was linked in the forum (http://www.pyregex.com/).

```
# Don't forget to import the library.
import re

# This opens the file for reading and reads all lines as a
# list of strings (called appointments).
dates = open("dates.csv", "r+")
appointments = dates.readlines()

# Printing them all before we do anything.
print("Here are all appointments, unprocessed:")
print(appointments)
```

[6] If you need a refresher about file input/output, see: https://www.pythonforbeginners.com/cheatsheet/python-file-handling

[7] https://docs.python.org/3/library/stdtypes.html#str.split

[8] https://docs.python.org/3/library/re.html#re.sub

```
# Here, the fun begins. Every app is one whole line in the original file.
for app in appointments:
    # Splits the line at the semicolons.
    separated = app.split(';')

    # Creates three _named capture groups_: month, day, and year.
    # The former two are one or two digits, the latter is four digits.
    american_date = r"(?P<month>\d{1,2})/(?P<day>\d{1,2})/(?P<year>\d{4})"

    # Establishes pattern to display. With \g<name> we can refer back
    # to the named capture groups from above.
    # (We know this from reading the documentation of re.sub() online)
    german_date = "\g<day>.\g<month>.\g<year>"

    # Actually execute the substitution of the new pattern in place of
    # the old and save the result into a new variable.
    new_date = re.sub(american_date,german_date,separated[1])

    # Print some useful information
    print("Don't forget! You have", separated[0], # Appointment type
                                "on the", new_date, # New European date
                                    "at", separated[2]) # Location
```

Assignment 6 (Regular Expressions 2 & HTML) – Given 08.12.2018, Due 14.12.2018

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 6.1 (Regular Expressions 3)

One of the best uses of a computer's enormous processing power is to have it filter quickly through large amounts of data that would otherwise take a human a long time to sift through. This is also often a task where regular expressions shine. 25pt

Along with this exercise, you will be supplied with a text file that contains the entire text of Lev Tolstoy's "War and peace"[9], slightly modified.[10] This will serve as our "corpus data" for this exercise.

Somewhere in this text (more than 500 kilowords), you know that there are a few e-mail adresses and a few hexadecimal colour codes (in a format like the following: #10FFAA). Write a python program that reads the file and uses regular expressions to find these addresses and colour codes. Afterwards, display the results with some explanatory text.

Note: Simply searching for "#" or "@" will not help you here, because since the data is sadly a bit "degraded", those characters are also interspersed a few hundred times at random intervals.

Solution:

```
import re

book = open('war-and-peace_modified.txt', 'r')
text = book.read()
pattern_c = r"#[0-9A-F]{6}"
results_c = re.findall(pattern_c,text)

print("The colour pattern appears", len(results_c), "times in the source.")
print(results_c)

pattern_e = r"[a-z]+\.?[a-z]+\@[a-z]+\.[a-z]+"
results_e = re.findall(pattern_e,text)

print("The email pattern appears", len(results_e), "times in the source.")
print(results_e)

book.close()
```

Problem 6.2 (HTML table)

In the lecture you saw the overview table for HTML below. 40pt

[9] As found on Project Gutenberg: `https://www.gutenberg.org` (currently not accessible from Germany due to copyright disputes)

[10] Found here: `https://kwarc.info/teaching/IWGS/materials/war-and-peace_modified.txt`

purpose	elements	purpose	elements
structure	`html`,`head`, `body`	metadata	`title`, `link`, `meta`
headings	`h1`, `h2`, ..., `h6`	paragraphs	`p`, `br`
lists	`ul`, `ol`, `dl`, ..., `li`	hyperlinks	`a`
multimedia	`img`, `video`, `audio`	tables	`table`, `th`, `tr`, `td`, ...
styling	`style`, `div`, `span`	old style	`b`, `u`, `tt`, `i`, ...
interaction	`script`	forms	`form`, `input`, `button`
Math	MathML (formulae)	interactive graphics	vector graphics (SVG) and `canvas` (2D bitmapped)

Make a HTML file `html-table.html` that re-creates this table in HTML. Note that the table heading is boldface and all of the HTML element names in the right column are in typewriter font (but the commata, ellipses, and explanations are not.)

Hint: You do not have to re-create the lines in the table. If you want to have (some kind of) border around the table cells, just add border="1" to the table element (or the tr elements).

Solution: Here's the source code for one way of creating this table. Most things are straightforward. First row uses `th` elements instead of `td` elements because these are the *table headers*.

```
<!DOCTYPE html>
<html>
<body>
<table border="1">
  <tr>
    <th>purpose</th>
    <th>elements</th>
    <th>purpose</th>
    <th>elements</th>
  </tr>
  <tr>
    <td>structurel</td>
    <td><tt>html, head, body</tt></td>
    <td>metadata</td>
    <td><tt>title,link,meta</tt></td>
  </tr>
  <tr>
    <td>headings</td>
    <td><tt>h1,h2,...,h6</tt></td>
    <td>paragraphs</td>
    <td><tt>p, br</tt></td>
  </tr>
  <tr>
    <td>lists</td>
    <td><tt>ul,ol,dl,...,li</tt></td>
    <td>hyperlinks</td>
    <td><tt>a</tt></td>
  </tr>
```

```
  <tr>
    <td>multimedia</td>
    <td><tt>img,video.audio</tt></td>
    <td>tables</td>
    <td><tt>table,th,tr,td,...</tt></td>
  </tr>
  <tr>
    <td>styling</td>
    <td><tt>style,div,span</tt></td>
    <td>old style</td>
    <td><tt>b,u,tt,i,...</tt></td>
  </tr>
  <tr>
    <td>interaction</td>
    <td><tt>script</tt></td>
    <td>forms</td>
    <td><tt>form,input,button</tt></td>
  </tr>
  <tr>
    <td>Math</td>
    <td>MathML (formulae)</td>
    <td>interactive graphics</td>
    <td>vector graphics (SVG) and <tt>canvas</tt> (2D bitmapped)</td>
  </tr>
</table>
</body>
</html>
```

Problem 6.3 (A Simple HTML Page)

Have a look at `https://www.izdigital.fau.de/efi-digitale-souveraenitaet/`. This page has header and footer parts (in blue) and two columns of text in between. The left one has the main text of the page (the page payload) and the right one some information about other pages on the same web site. 35pt

Make a simple web page from the payload text and the page heading "EFI-Förderung für das Forschungsprojekt „Diskurse und Praktiken einer digitalen Souveränität"".

1. Download the file `https://kwarc.info/teaching/IWGS/materials/efi.txt`, save it, and rename it to `efi.html`.

2. With the HTML tags we have introduced in the lecture mark up all structural parts: paragraphs, itemized lists, hyperlinks (Hint: you can obtain the link target by right-clicking on the hyperlink and selecting "Copy Link Address". You only need to mark up five links total.)

3. Load your `.html` file into a browser of your choice (this acts as the HTML document viewer) and export the contents to PDF (call the file `efi.pdf`).

4. Use the HTML checker at `https://validator.w3.org/#validate_by_upload` to see what it thinks of your HTML. Correct your errors reported there (as much as reasonable). Briefly discuss what your experience has been with this tool.

Submit `efi.html`, `efi.pdf`, and your discussion from 4.

Solution: On our website[11] you can find `efi.html` a fully marked up HTML version of the site that clears the checker without errors or warnings.

The most problematic part of this exercise was probably to get a grip on special German characters in HTML. These need to be replaced by special sequences in HTML[12] or they will be displayed in a messed-up way.

[11] `https://kwarc.info/teaching/IWGS/materials/`

[12] Which sequence goes where can be looked up here: `https://wiki.selfhtml.org/wiki/Referenz:HTML/Zeichenreferenz`

Assignment 7 (HTML 2 & Regular Expressions 3) – Given 15.12.2018, Due 21.12.2018

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 7.1 (Simple HTML Form) 30pt

For this exercise, you will construct a very simple HTML page with a basic form. Suppose you want to establish a basic pizza delivery service only for **FAU** staff and students. It is your task to make the first version of the website for the "front-end" (that is, the user-facing part of the application).

Create a `.html` file[13] with a title, a heading, a paragraph or so of descriptive text and a `<form>`-element that contains the following inputs:

- a text input field for people to enter their name,
- a dropdown menu with (at least three) FAU-related addresses,
- (at least three) radio buttons labeled with different pizza options (for the moment, we only allow one pizza to be ordered at a time).
- a form-submit button.

When the submit button is clicked by the user, they should be redirected to a second HTML page (hand this in, too, in a separate file), that tells the user their order has been received. Use the form `action` attribute to accomplish this. This second page does *not* need to use the data from the form.

Solution: You can easily find and adapt HTML snippets in the linked source. For the text input field, we use `<input type="text" ...>`, for the dropwdown, we use a `<select>` element with `<option>`s inside of it and for the radio buttons we use `<input type="radio" ...>`. The radio buttons all need to have the same name so only one of them can be selected at any given time. The button can be created with a `<input type="submit"`, all of which goes inside a `<form>` element.

One possible `.html` file that fits the bill would be this one:

```
<!DOCTYPE html>
<html>
<head>
    <title>FAU Pizza Delivery</title>
    <link rel="stylesheet" href="styles.css">
</head>
```

[13] If you need a refresher: there is excellent documentation on how the basics work at `https://www.w3schools.com/html/html_intro.asp` and related pages.

```
<body>

<h1>FAU Pizza Delivery</h1>

<p>Welcome to the FAU-internal pizza delivery service!
<br>Please enter your name, your location and select which pizza you would like delivered.</p>

<form action="submitted.html">
  <b>Name:</b> <input type="text" name="name" value="Vorname Nachname">
  <br><br>
  <b>Address:</b>
  <select name="address">
<option value="kollegienhaus">Kollegienhaus</option>
<option value="bismarckstrasse">Bismarckstrasse</option>
<option value="martensstrasse">Hochhaus Martensstrasse</option>
  </select>
  <br><br>
  <b>Selection:</b><br>
  <input type="radio" name="pizza" value="margherita" checked> Pizza Margherita<br>
  <input type="radio" name="pizza" value="salame"> Pizza Salame<br>
  <input type="radio" name="pizza" value="vegetaria"> Pizza Vegetaria
  <br><br>
  <input type="submit" value="Submit">
</form>

</body>
</html>
```

With this being one possible source code for `submitted.html` that is being redirected to when the submit button is clicked:

```
<!DOCTYPE html>
<html>
<head>
    <title>FAU Pizza Delivery</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>

<h1>FAU Pizza Delivery</h1>

<p>Welcome to the FAU-internal pizza delivery service!
<br/>Your order has been received!</p>
</body>
</html>
```

Problem 7.2 (Simple CSS)

It is a well-known fact that nobody likes to buy from a pizza place that only uses plain 20pt

HTML on their website. So now, we will improve upon the website from Problem 7.1.

Create an external style sheet (in a CSS-file called `styles.css`) to change the look of your website. You can load this style sheet by placing the following `head`-element into your website's `html`-element:

```
<head>
    <link rel="stylesheet" href="styles.css">
</head>
```

You can make this style sheet as elaborate as you like. However, at least the following style changes should be implemented by your style sheet:

- Center the heading.
- Give the `<body>` of your website a `background-color`.
- Set the `font-family` of all text to "Verdana".
- Set the font size of your descriptive text to 14.

Solution: Here is one CSS files that can fill the role of `styles.css`:

```
body {
  background-color: #669999;
}

h1 {
  text-align: center;
}

p {
  font-family: verdana;
  font-size: 14pt;
}
```

Problem 7.3 (Regex Parsing)

Suppose that you are now working on the python "back-end" (that is, the part of the software that is managing and manipulating the data) of your **FAU**-internal pizza delivery sercive from Problem 7.1. 50pt

Say you have a log file where in each line there is a percent-encoded[14] HTML POST request to your website. Each of them encodes the *name*, *address* and *pizza choice* of one order, like in the following examples:

```
POST name%3DTheo+McTestPerson%26address%3Dkollegienhaus%26pizza%3Dsalame
POST name%3DMax+Musterfrau%26address%3Dkollegienhaus%26pizza%3Dvegetaria
POST name%3DBea+Beispielname%26address%3Dmartensstrasse%26pizza%3Dsalame
...
```

[14] See: `https://en.wikipedia.org/wiki/Percent-encoding`

Such a file is also being provided along with this exercise.[15] Write a program that first reads that file and creates a list of python dictionaries (one for each order, with the keys "name", "address" and "pizza") out of the included data.[16] Use regular expressions to find the corresponding values in the data.

The program should then do the following:

- Your program needs to compute (and print) what sorts of pizzas were ordered and how many of each are needed in total.
- Your program should also print all adresses that the delivery driver needs to go to.
- Lastly, your program should compute and display the total amount of money that you would expect to be paid for this delivery (you can assign an arbitrary price to each variety of pizza for this exercise).

Solution: This exercise was designed to be a little less direct on how the task should be accomplished and only specified *what* should happen, not necessarily *how* to program that.

The regular expression should not be too difficult at this point, since everything used here has been mentioned and used multiple times before.

The relevant parts are how to keep track of how many pizzas of a certain type were ordered, for example. For this, we can use a dictionary, that has pizza types (strings) as keys and integers as values. We loop over all orders and increase / insert the value for the corresponging key in this big order dictionary.

Similarly, we can keep a list of all addresses that we need to send a driver to.

```
import re

# Read logfile
logfile = open('console.log', 'r')
orders_raw = logfile.read()

# Close open file, we don't need it anymore.
logfile.close()

# Use regular expression to match on all orders
orders = re.findall("POST name%3D(.+)%26address%3D(.+)%26pizza%3D(.+)", orders_raw)

# Create an empty list to append dicts to.
order_dicts = []

# Loop over all orders we matched on
for order in orders:
    # empty dictionary
    nu_dict = {}
```

[15] Found here: `https://kwarc.info/teaching/IWGS/materials/console.log`

[16] You can read up on how to create and/or add key-value-pairs to dictionaries in a program here: `https://www.w3schools.com/python/python_dictionaries.asp`

```
    # Names still have "+" instead of space, substitute the two.
    spaced_name = re.sub("\+", " ", order[0])

    # Add values from the order.
    nu_dict["name"] = spaced_name
    nu_dict["address"] = order[1]
    nu_dict["pizza"] = order[2]

    order_dicts.append(nu_dict)

# PART 1

pizza_dict = {}

for d in order_dicts:
    # The type of pizza in this order
    ptype = d["pizza"]

    # Increment counter in dictionary
    # (or set to 1 if previously not in dictionary)
    if (ptype in pizza_dict):
        pizza_dict[ptype] = pizza_dict[ptype] + 1
    else:
        pizza_dict[ptype] = 1

for pizzatype in pizza_dict:
    print("There were", pizza_dict[pizzatype], "orders of", pizzatype)

# PART 2

addr_list = []

# Add all addresses (that are not yet in there) to the list
for d in order_dicts:
    if d["address"] not in addr_list:
        addr_list.append(d["address"])

print("")
print("Driver needs to go to these addresses:")
for addr in addr_list:
    print(addr)

# PART 3

# In this silly example, the price of the pizza is the length
# of its name, in euros.
def price(pizzatype):
    return len(pizzatype)
```

```
# We can re-use our results from Part 1!

total = 0
for pizzatype in pizza_dict:
    amount = pizza_dict[pizzatype] # how often was this pizza ordered?
    one_price = price(pizzatype) # What does one pizza cost?
    total += amount * one_price

print("")
print("Total amount of money due:", total, "Euros")
```

Assignment 8 (Project Design & Prototyping) – Given 22.12.2018, Due 11.01.2019

Note: We realize that the workload of the last weeks was high, especially for those of you currently also attending GDI. Therefore, we would like to emphasize that this project is *purely optional*. If you feel like you still have trouble understanding some of the subjects of the lecture, we recommend that you don't participate in the project, and instead try to tighten your understanding of the topics of the last weeks.

Both Jonas and Philipp are more than willing to help you grasp the presented concepts. We strongly advise you to repeat the previous exercise sheets and we are happy to accept solutions to these assignments and give you feedback for them.

We know that programming can be hard and that learning a large amount of new concepts in rapid succession can feel overwhelming, but we are sure that if you go back to these exercise sheets now you will realize just how much you've learned so far. Please make use of this opportunity.

Again: This project is intended for those who can't get enough of programming. You will do just fine in the exam, if you did not complete the project, but instead have a solid understanding of the concepts presented in the lecture.

Problem 8.1 (Project 02)

Write a short summary (bullet points) of the core interactions the user is intended to perform on your project application. In which order are they expected to use the interface elements (checkboxes, buttons, dropdown menus, ...)? 40pt

How must these elements be wired internally? If the user interacts with a particular element, what is internally supposed to happen? What data elements will be shown to the user? Knowing these expected interactions, how is your data supposed to be organized in order to provide the wanted functionality? Which data structures (lists, dictionaries, lists of dictionaries, ...) do you expect to be using? Where does the data come from (files on the hard drive like CSV tables, from other websites like Wikipedia, ...)?

Solution: *No example solution for this problem.*

Problem 8.2 (Project 03 (Paper Prototype))

With the knowledge of your desired interaction scheme (from Problem 8.1), draw (on a sheet of paper) the basic layout of your application. An very simplistic example of what this could look like is shown on page 59 of the course notes. 20pt

Scan or photograph your design for the submission.

Solution: *No example solution for this problem.*

Problem 8.3 (Project 04 (HTML Prototype))

Create a first prototype of your design as a HTML/CSS website (whether or not your application was designed as a website, since this is only a *prototype*). 40pt

Try to position each element as planned in Problem 8.2. Since we are only working on the layout of the page in this exercise, your buttons etc. don't have to work yet.

Solution: *No example solution for this problem.*

The whole IWGS team wishes all of you happy holidays!

Assignment 9 (Trees in Python and XML) – Given 12.01.2019, Due 18.01.2019

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 9.1 (Trees in Python & Recursion) 80pt

During the lecture, you learned about the very important data structure of *trees*. In this exercise we will be taking a closer look at *binary* trees (trees where every non-empty node has exactly two children) of integers.

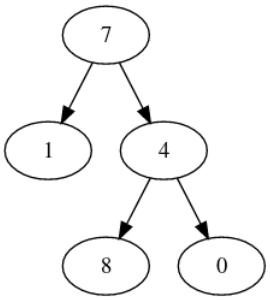
One way of implementing trees in Python is *nested dictionaries*. Every node in the tree is either the empty dictionary ({}, this is called a *leaf* of the tree) or a dictionary with the keys "value" (which for this exercise will be an integer), "left" and "right". The latter two are both dictionaries that are again either empty or trees with a value and two children.

You can find an example tree constructed in this manner in the code snippet below and a visualization of the same tree below.

```
# Example for a tree as nested dictionaries.
treeA = {"value":1, "left":{}, "right":{}}
treeB = {"value":8, "left":{}, "right":{}}
treeC = {"value":0, "left":{}, "right":{}}

treeD = {
  "value" : 4,
  "left" : treeB, "right" : treeC
}

exampleTree = {
  "value" : 7,
  "left" : treeA, "right" : treeD
}
```



A visual representation of the tree encoded as dictionaries on the left.

Write a Python function called treeMinimum that takes a (non-empty) tree as input (you can take exampleTree from above as a test case, but it needs to work for all trees constructed this way) and finds the *smallest* integer that any node in the tree carries. For example, for the tree above, your function should return 0.

Hint: To do this, you will have to understand and use *recursion*, another very important concept in informatics[17] (read up on this first!). This means that, from within treeMinimum, you will need to call treeMinimum again on the children of the node you're currently inspecting. This will give you the smallest integers that are anywhere below the current node. The fact that you are

[17]For more on this, see: `https://en.wikipedia.org/wiki/Recursion_(computer_science)` or `https://www.python-kurs.eu/rekursive_funktionen.php`

solving the larger problem (minimum of the larger tree) by calling the same function on smaller problems (minimum of the left and right branches respectively) makes this *recursive*.

You might also find the min-function[18] helpful.

Solution: At the heart of every recursive solution to a problem lies the central idea of making the problem *smaller*, by splitting it into parts (divide and conquer!). This means that we only need to concern ourselves with two aspects of the problem.

- The simplest case that cannot be reduced further.
  (In this case: *The smallest node in a tree that consists only of one node with no children is the node itself.*)
- The complex case of a problem, but with all the subproblems already solved.
  (In this case: *The smallest node in a tree that consists of a root node* with *children is the minimum of the node itself, the smallest node in the left child tree and the smallest node in the right child tree.*)

This is enough to solve even very large problems because at every stage, the problem is either the simplest it can be (node with no children) or a combination of subproblems (node with children). So we can assume the subproblems already solved because we can apply the function *recursively* to the subproblems and it will work the same there, too. At some point we will reach the simplest stage (where we hopefully know directly what to do) and from there on up we can use the partial solutions to solve the larger problems.

```
# Define a function that takes a tree (dictionary) as input.
def treeMinimum(tree):
  # We create a list with all values that we want to compare.
  # That is: 1. The value this node carries.
  # 2. The minimum of the left branch (if that is not empty).
  # 3. The minimum of the right branch (if that is not empty).
  compare = [tree["value"]]

  # If this node has a non-empty left child, we add the minimum of
  # that branch (which we find _recursively_) to our comparison list.
  if (tree["left"] != {}):
    # Since the left child of a tree is also a tree, we can call this
    # function again.
    # However, this only works if the child is not the empty dictionary,
    # that's why we put this call into this "if" block.
    leftMinimum = treeMinimum(tree["left"])

    # The append function adds an element to the end of our list.
    compare.append(leftMinimum)

  # Analogous to the left case, but with the right branch.
  if (tree["right"] != {}):
    rightMinimum = treeMinimum(tree["right"])
```

[18] This function finds the smallest element in a list. See: `https://docs.python.org/3/library/functions.html#min`

```
    compare.append(rightMinimum)

  # Find the minimum of all values that are relevant at this level.
  minimum = min(compare)

  # Return the minimum we found.
  return minimum

print("The smallest integer in this tree is", treeMinimum(exampleTree))
```

Problem 9.2 (XML)

In this exercise, we will discuss the XML language family. Please answer the following questions (at most a few sentences each): 20pt

1. What is the difference between XML and HTML?
2. What roles do trees play for those two?
3. Name at least three uses of XML.

Give a short example of valid XML code that you have written yourself. Also give a small example of *incorrect* XML and explain why exactly your example is incorrect.

Solution: *No example solution for this problem.*

Assignment 10 (Generating HTML with Python) – Given 18.01.2019, Due 25.01.2019

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 10.1 (Generating HTML elements)

One of the biggest advantages in programming is *automation*, recognising structured tasks that come up a lot and replacing human effort with computation. In these exercises we will try and automate the "boring" parts of generating simple websites in HTML. 25pt

First, write two functions, wrapH1 and wrapP, that take one argument and *return* (not to be confused with "print"!) a string. The return string should be an opening tag (`<h1>` and `<p>` tags respectively), followed by the argument to the function, and then the matching closing tag.

Solution: These functions are pretty easy in themselves, once you know how to define a function. There is only minimal string concatenation required.

```
def wrapP(payload):
    return "<p>" + payload + "</p>"

def wrapH1(payload):
    return "<h1>" + payload + "</h1>\n"
```

Problem 10.2 (Generating website skeleton)

Next, write a function wrapQuickFacts that takes 5 string arguments and returns a string describing a HTML table[19] listing these arguments under the categories "Name", "Job Title", "Date of Birth", "Email", and "Website". 25pt

Finally, write a python function wrapSkeleton that analogous to those in Problem 10.1, return the general structure of a basic HTML page[20] as a string. The function should also take a string as an argument that is inserted between the opening and closing `<body>` tags in the returned string.

Solution: These functions also consist mostly of string building. The key aspects to get right are the function definition, correct HTML and putting the arguments into the right places.

```
def wrapQuickFacts(name, job, date, mail, site):
    html = "<table>\n"
    html += "<tr><td>Name:</td><td>" + name + "</td><tr>\n"
    html += "<tr><td>Job Title:</td><td>" + job + "</td><tr>\n"
    html += "<tr><td>Date of Birth:</td><td>" + date + "</td><tr>\n"
```

[19] If you need a refresher on this, you can find this structure here: `https://www.w3schools.com/html/html_tables.asp`

[20] See also: `https://www.w3schools.com/html/html_intro.asp`

```
    html += "<tr><td>Email:</td><td>" + mail + "</td><tr>\n"
    html += "<tr><td>Website:</td><td>" + site + "</td><tr>\n"
    html += "</table>\n"
    return html
```

The wrapSkeleton function is even simpler.

```
def wrapSkeleton(payload):
    html = "<html>\n"
    html += " <body>" + payload + "</body>\n"
    html += "</html>"
    return html
```

Problem 10.3 (Generating complete websites)

After we have solved the smaller problems, it is now time to combine the solutions into a (slightly) bigger program. 50pt

Using your results from Problem 10.1 and Problem 10.2, write a python function generateWebsite that, given a dictionary with appropriate data[21] as input, generates (i.e. returns the HTML string that describes) the complete website including a heading, the table and a paragraph of flavour text and saves it into a `.html` file.

Generate one of these websites for all entires in peopleList using the functions you wrote.

Solution:

```
def generateWebsite(person):
    filename = person["name"] + ".html"
    table = wrapQuickFacts(person["name"],person["job"],person["date"],person["mail"],person["site"])
    complete = wrapSkeleton(wrapH1(person["name"]) + table + wrapP(person["text"]))

    with open(filename,"w") as website:
        website.write(complete)

for person in people.peopleList:
    generateWebsite(person)
```

---

[21] You can find a file with example data here: `https://kwarc.info/teaching/IWGS/materials/people.py` You can either copy-paste these or have the file next to yours and use **import** people in your file to be able to use people.peopleList.

Assignment 11 (Web-Applications with Bottle) – Given 26.01.2019, Due 01.02.2019

Note: Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, ask a fellow student, your tutor or consult the lecture materials at `https://kwarc.info/teaching/IWGS/`.

Problem 11.1 (Hello WebApp World) 10pt

In these exercises, we will take a closer look at web-applications, templating and HTML routing. Concretely, we will be using the Bottle framework[22], as demonstrated in the lecture.

Set up a web-application on your `pythonAnywhere.com`-account[23] using the Bottle framework. `pythonAnywhere.com` has a quickstart option for bottle.

Set up the following routes (pairs of URLs and python functions that return strings):

- A client navigating to the root directory of your webapp (`"/"`) should receive a standard `"Hello World"` message.
- A client navigating to `"/hello/<name>"` should find a greeting message personalised with the name given in the URL (`"/hello/Philipp"` greets Philipp, `"/hello/Jonas"` greets Jonas, ...).

  Have at least one name (your choice) be treated differently than all others (for example: all names get a nice message by default, but the name `"GrumpyCat"` gets an annoyed message).

Solution: A route in this context is a pair of an URL and a python function returning a string. We can set up a route in `bottle` with the `@route(url)` *decorator* in our source file. Directly below that we define a function that returns an appropriate string.

For the first part of this exercise, this is easily done:

```
@route("/")
def genericHello():
    return "Hello World"
```

We can also include a placeholder, like `<name>` in the url that we give to the route decorator. This can then function as an argument to the function we define for the route.

```
@route("/hello/<name>")
def specificHello(name):
    if name == "GrumpyCat":
        return template('Oh no {{name}}, it\'s you. Get lost!', name=name)
```

[22] See the documentation of bottle for reference: `https://bottlepy.org/docs/dev/tutorial.html`

[23] See this help page for instructions: `https://help.pythonanywhere.com/pages/WebAppBasics/`

```
    else:
        return template('Hello {{name}}, how are you?', name=name)
```

Problem 11.2 (Routing a HTML form)

In the following exercises, we want to build a small, but complete (!) web application where users can submit reviews for media (books, movies, ...) that get saved into a "database" and can be viewed later. A lot of these exercises will ask for HTML or python code that is similar to previous exercises. The challenge is to integrate the familiar code into the new context of web-applications and the bottle framework. 20pt

Add a `"/submit"` route to your web app that delivers a HTML form. The form should at least have `input` elements for a title (text), a synopsis (text) and a rating from 1 to 5 (number or radio buttons).

When the submit button (which also needs to be included in the form) is pressed, the form should redirect the user to the `"/submitted"` route (see Problem 11.3) via the `action` attribute. Make sure that the method used for this is a GET request (how can you specify this?).

Solution:

This exercise is largely similar to a previous exercise, with the only difference being that the HTML source code needs to be returned as a string for the appropriate route here.

```
@route("/submit")
def submitForm():
    page = """
    <h1>Media Review Submission:</h1>
    <form action="/submitted" method="get">
    Media Title:<br>
    <input type="text" name="title"><br><br>
    One-sentence-synopsis:<br>
    <input type="text" name="synopsis"><br><br>
    Rating (between 1 and 5):<br>
    <input type="number" name="rating" value="5" min="1" max="5"><br><br>
    <input type="submit" value="Submit">
    </form>
    """
    return page
```

Problem 11.3 (HTML GET Requests)

Now, add a route specifically for GET requests at `"/submitted"` (the target of your submit-redirect from Problem 11.2). Since we're dealing with a GET request, the information submitted through the form will be encoded in the URL. 40pt

The corresponding function should read the title, synopsis and rating from the HTML request (see the bottle documentation or the lecture materials for examples) and append

them to a file[24] called `database.txt`[25].

You can append one line of text to the file per entry in the database, with the title, synopsis and rating seperated by semicolons, for example.

Solution: The actual string being returned by this route is only marginally interesting. Were mainly interested in the side effect of appending an entry to the database file.

To only accept GET-requests at this URL, we use the `get`-decorator, otherwise, all remains the same.

We can use a `with`-statement to open the file `database.txt` into the variable db for writing (appending only in this case). This is equivalent to file handling in earlier exercises but is considered prettier code.

We can use request.GET.xxx to get the `xxx` attribute of our GET request (that was encoded in the URL, like seen in earlier exercises).

```
@get("/submitted")
def submitted():
    with open("database.txt", "a") as db:
        theTitle = request.GET.title.strip()
        theSynopsis = request.GET.synopsis.strip()
        theRating = request.GET.rating.strip()
        txt = template("{{t}} ({{r}}/5): {{s}}\n",t=theTitle,s=theSynopsis,r=theRating)
        db.write(txt)

    txt = "Thank you! Your entry has been submitted.<br>"
    nu = "<a href=\"/submit\">Submit another one</a> or "
    dat = "<a href=\"/database\">view the database</a>."

    return txt + nu + dat
```

Problem 11.4 (Displaying the database)

Finally, add a `"/database"` route to your web app that reads the aforementioned database file (`database.txt`) and displays its contents as a HTML page. This page should contain a heading and an unordered list (the `<ul>` element), in which each entry in the database (= line in the file) is one list item (`<li>` element). 30pt

Solution: This exercise is mostly combining multiple aspects of previous exercises. We can read `database.txt` as normal and build up a string via a for loop over all lines in the database file.

```
@route("/database")
def displayDB():
    disp = "<h1>ALL CURRENT ENTRIES:</h1>\n<ul>"
    with open("database.txt") as db:
        count = 0
        for line in db.readlines():
```

[24] Even though the function must ultimately *return* a string from which a HTML page is constructed, it can write to a file before doing so as a side effect.

[25] This file will appear next to your other files in your `pythonAnywhere` directory. It is enough to simply append to the file, python will create the file if it does not exist yet.

```
            disp += template("<li>Entry {{n}}: {{e}}</li>", n=count, e=line)
            count += 1

    disp += "</ul>"
    return disp
```