1 Meet Python

1.1 Questions

▸ Q: What do the following terms mean: programming language, scripting language, compiled language, interpreted language?
A: The four terms have the following meaning:

- Programming language: Used to build large and complex software applications. C++, Java, Fortran and Python are examples for a programming languages.
- Scripting language: Used for smaller tasks often to automate repetitive tasks. AWK, Bash, Perl, sed and Python are examples of languages used for scripting.
- Compiled language: Source code written in a compiled language has to be compiled using a compiler to create an executable file. Compilation means the translation of human-readable source code into machine-code understandable by the computer. Executable languages are better performing than interpreted languages since the source code is translated into machine-code before runtime. Furthermore compilers can optimize the machine-code and compilation errors can help to debug code, although they can be sometimes cryptic (e.g. Segmentation Fault in C++). Examples for compiled languages are C++ and Fortran.
- Interpreted language: Source code written in an interpreted language is translated during runtime, line by line, into machine-code. Without the context of the rest of the source code, the interpreter cannot optimize the machine-code as well as the compiler. An example for an interpreted language is Python.

▸ Q: What is Python?
A: Python is a programming language with the following characteristics:

- High-level: Uses strong abstraction from details of a computer. In contrast to low-level programming languages which allow operation on the memory, high-level languages do not concern themselves with the hardware details.
- General purpose: Build software for a wide variety of application domains.
- Dynamically typed: Types of variables are checked during runtime to ensure type safety, e.g. the addition of an integer with a string is caught as invalid operation.
- Garbage collected: Automatic memory management, e.g. memory allocated for arrays during a procedure call is released after its termination.
- Supports multiple programming paradigms: procedural (modularize code with procedures), object-oriented (use objects with methods and attributes), functional (programs are constructed by combining pure functions without side-effects).

▸ Q: Why is Python so popular?
A: Python allows to write clean code, is modular and extendable.

▸ Q: What is a typed language and what are types in Python?
A: A typed language is a programming language in which the programmer specifies the data type of variables such as integer, float, character, strings or boolean. One distinguishes between statically (type checked during compilation, e.g. C++) and dynamically (type checked during runtime, e.g. Python) typed languages. Another way to distinguish languages is by strong (fixed rules for type conversions) and weak (loose rules for type conversion) typing.
The main datatypes in Python are:

- Integer, Float, Complex:
  - Integer: Discrete numbers in the range $[-2^{-31}, 2^{31}-1]$ for a 32 bit system and $[-2^{-63}, 2^{63}-1]$ for a 64 bit system.
  - Float: Continuous numbers represented by a sign s, a mantissa m and an exponent x giving $f = (-1)^s \cdot m \cdot 2^x$. Numbers are stored in units of 32 bit (single precision), 64 bit (double precision) or 32 bit (extended precision). In Python, double precision is used with 1 bit for the sign,

52 bit for the mantissa and 11 bit for the exponent. Due to the finite precision, numbers which cannot be exactly represented by powers of 2 introduce rounding errors, e.g. the comparison `0.1+0.1+0.1 == 0.3` gives `False`. Hence when using floats, one should not test for equality, but check if the absolute values of differences are smaller than some tolerance. Furthermore numbers should be scaled such that they lie in a similar range, otherwise rounding errors emerge when adding large and small values.

Operation	Call
Initialization	`x=10`, `x=10.0=1e1`, `x=10+1j`
Constructor/Conversion	`int(x)`, `float(x)`, `complex(re,im)`
Unary operations	`+x`, `-x`, `abs(x)`
Binary operations	`x+y`, `x-y`, `x*y`, `x/y`, `x//y` [floor(x/y)], `x%y` (remainder of x/y), `pow(x,y)` = `x**y` = x^y
Comparison	`x>y`, `x<y`, `x==y`, `x!=y`, `x>=y`, `x<=y`
Complex conjugation	`c.conjugate()`
Bitwise operations of Integer	`x|y` (x or y), `x^y` (x xor y), `x&y` (x and y), `x<<n` (shift x left by n bits), `x>>n` (shift x right by n bits), `~x` (invert bits)

- String:
  - Zero-based indexing of elements
  - Elements are immutable
  - f-string: Formatted strings where variables can be expanded by putting them inside curly braces, e.g. val = 17 →f'{val}' = '17'
  - r-string: Raw string literals where a backslash is treated as literal character, i.e. it does not have to be escaped to avoid expansion, e.g. `r'Hi\n␣Jan'` →`'Hi\\n␣Jan'`

Operation	Input	Output
Initialization	`s1='apple␣bAnana␣'`, `s2='pair.␣orange'`, `s3='Test\_'`, `s4='12'`	–
Constuctor/Conversion	`str()`	–
Concatenation	`s1+s2`	`'apple␣bAnana␣pair.␣orange'`
Repetition	`2*s3`	`'Test_Test_'`
Indexing	`s1[2]`	`'p'`
Slicing	`s1[6:9]`	`'bAn'`
Capitalize string	`s2.capitalize()`	`'Pair.␣orange'`
Count substrings	`s1.count('a')`	`3`
Find substring	`s1.find('bA')`	`6`
Get index of substring	`s1.index('bA')`	`6`
All lower case	`s1.lower()`	`'apple␣banana␣'`
Replace substring	`s1.replace('a','u')`	`'upple␣bAnunu␣'`
Split at separator	`s1.split('␣')`	`['apple','bAnana','']`
Concatenate any number of strings (insert calling string between elements of list of strings)	`'-'.split(s1.split())`	`'apple-bAnana'`
Remove leading, trailing whitespace	`s1.strip()`	`'apple␣bAnana'`
All upper case	`s1.upper()`	`'APPLE␣BANANA␣'`
Fill with leading zeros	`s4.zfill(5)`	`'00012'`

- Boolean:
  - Values: `True`, `False`
  - Logical operations: `x and y`, `x or y`, `not x`

Q: What is a variable?
A: A variable is a symbol assigned with a value. It is associated with a space in memory where the value is save.

Q: How is code organized in Python?
A: Code in Python is organized using functions and modules. While functions combine code to fulfill tasks, modules are essentially Python files and usually consist of a collection of functions.

Q: What are function parameters and return values?
A: Function parameters are values passed to a function while return values are values returned by a function. They can be unnamed arguments or named keyword arguments. In case the number of unnamed and named keyword arguments is unknown, list and dictionary unpacking is used as `def function(*args,**kwargs)`, where `args` contains the list of unnamed arguments and `kwargs` the dictionary of named keyword arguments.

Q: Which control structures are there in Python?
A: There are three types of control structures in Python [Unk23b]:

- Sequential: Normal sequential execution
- Selection: if, if-else, nested if, if-elif-else, match-case statements
- Repetition: for-, while-loops

Q: When does one use which control structure?
A: The sequential control structure is used during regular execution. Selection control structures are used when a branching decision determines the progress. Repetition control structures are used in case a task has to be repeated multiple times.

Q: Which rules-of-thumb are there for using control structures?
A: Best practices for selection control structures:

- Use meaningful variable names to enhance code readability
- Indent code blocks consistently to improve code structure
- Add comments to explain complex conditions or code logic
- Avoid excessive nesting of if-statements to keep code clean and readable

Best practices for repetition control structures:

- Use meaningful variable names to enhance code readability
- Avoid infinite loops by ensuring the loop condition will eventually become false
- Minimize the use of nested loops to maintain code clarity and performance
- Use loop control statements (continue, break) judiciously and ensure they improve the codes logic

Q: What are the most common data structures in Python? What are their characteristics?
A: The most common data structures in Python [San22] are:

- List: Dynamic, mutable arrays which hold an ordered collection of items.
  Pro:
  - easiest way to store collection of relation objects
  - easy to modify by adding, removing and changing elements
  - useful to create nested data structures like a list of lists/dictionaries

  Contra:
  - slow when performing arithmetic operations on elements →use numpy arrays
  - use more disk space because of features

  Construction and useful methods:

Method	Input	Output	lst
Creation	`lst = [1,2,3,1]` `lst = list((1,2,3,1))`	–	`[1,2,3,1]`
Length	`len(lst)`	4	`lst`
Append	`lst.append(4)`	–	`[1,2,3,1,4]`
Count occurences	`lst.count(1)`	2	`lst`
First index of value	`lst.index(1)`	0	`lst`
Insert at index the value	`lst.insert(1,5)`	–	`[1,5,2,3,1]`
Pop last value	`lst.pop()`	1	`[1,2,3]`
Remove first occurence of value	`lst.remove(1)`	–	`[2,3,1]`

Reverse	`lst.reverse()`	–	`[1,3,2,1]`
Sort	`lst.sort()`	–	`[1,1,2,3]`
Clear	`lst.clear()`	–	`[]`

List comprehension: `lst = [x*x for x in range(4)] = [0,1,4,9]`

- Dictionary: Dynamic, mutable data structures that contain a collection of keys and associated values.
  Pro:
  - make code easy to read when needing key:value pairs
  - fast access to values using keys

  Contra:
  - need a lot of space →avoid when handling a large amount of data

  Construction and useful methods:

Method	Input	Output	dct
Creation	`dct = {'a':1,'b':2}` `dct = dict([['a',1], ['b',2]])`	–	`{'a':1,'b':2}`
Length	`len(dct)`	`3`	`dct`
Get	`dct.get('b')`	`2`	`dct`
Add element	`dct['c'] = 3`	–	`{'a':1,'b':2,'c':3}`
Show items	`dct.items()`	`dict_items([('a', 1), ('b', 2)])`	`dct`
Show keys	`dct.keys()`	`dict_keys(['a', 'b'])`	`dct`
Show values	`dct.values()`	`dict_values([1, 2])`	`dct`
Pop by key	`dct.pop('a')`	`1`	`{'b':2}`
Pop item added last	`dct.popitem()`	`('b', 2)`	`{'a':1}`
Clear	`dct.clear()`	–	`{}`

  The difference between requesting an entry by key (e.g. `dct['a']`) and the `get` method is, that if the key is not in the dictionary, the direct request throws an error, while the get method does not return anything. Using the defaultdict class in the collections module, each call to a non-existing key creates an entry with this key and a value determined by the `default_factory` attribute. Dictionary comprehension: `dct = {x:x$\ast$x for x in range(4)} = {0:0,1:2,2:4,3:9}`
- Tuple: Ordered, immutable arrays
  Pro:
  - immutable, i.e. accidental changes are not possible
  - can be used as dictionary keys

  Contra:
  - cannot be used when working with modifiable objects →use lists
  - cannot be copied
  - occupy more memory than lists

  Construction and useful methods:

Method	Input	Output	tpl
Creation	`tpl = (1,2,3,1)` `tpl = tuple((1,2,3,1))`	–	`(1,2,3,1)`
Length	`len(tpl)`	`4`	`tpl`
Count occurences	`tpl.count(1)`	`2`	`tpl`
First index of value	`tpl.index(1)`	`0`	`tpl`

Tuple comprehension: `tpl = tuple(x*x for x in range(4)) = (0,1,4,9)`

- Set: Dynamic, mutable collection of immutable, unique elements.
  Pro:
  - can perform unique operations (union, intersection) on them
  - much faster than lists to check if an element is contained in a set

  Contra:
  - intrinsically unordered
  - cannot change elements by indexing

Method	Input	Output	st1	st2
Creation	`st1 = {1,2,3,5}`	–		
	`st2 = set((2,3,4))`			
Length	`len(st)`	`4`	st	
Add	`st1.add(7)`	–	`{1,2,3,5,7}`	st2
Difference	`st1.difference(st2)`	`{1,5}`	st1	st2
Remove element	`st1.discard(5)`	–	`{1,2,3}`	st2
Intersection	`st1.intersection(st2)`	`{2,3}`	st1	st2
Check if disjoint	`st1.isdisjoint(st2)`	`False`	st1	st2
Check if subset	`st1.issubset(st2)`	`False`	st1	st2
Check if superset	`st1.issuperset(st2)`	`False`	st1	st2
Pop arbitrary element	`st1.pop()`	`1`	`{2,3,5}`	st2
Remove member	`st1.remove(5)`	–	`{1,2,3}`	st2
Symmetric difference	`st1.symmetric_difference(st2)`	`{1,4,5}`	st1	st2
Union	`st1.union(st2)`	`{1,2,3,4,5}`	st1	st2
Update	`st1.update(st2)`	–	`{1,2,3,4,5}`	st2
Clear	`st1.clear()`	–	`{}`	st2

For difference, intersection, symmetric difference and union there are methods for saving the results in the calling set (here: `st1`): `difference_update`, `intersection_update`, `symmetric_difference_update` and `update`. While `remove` throws an error when trying to remove a non-member of the set, `discard` stays silent and does not change the set.

Other, less commonly used data structures are stacks (last-in, first-out sequences →list), queues (first-in, first out sequences →collections.deque), numpy arrays (used to perform mathematical operations on arrays) or pandas DataFrames (data in tabular form grouped into rows, columns and labeled by indices) and Series (1D array). Notice that indexing in Python starts at zero.

Note that lists, tuples and sets can be converted into one another using constructors. Strings can be decomposed into their characters using the list and tuple contructors, while a set filters out doubled letters.

Q: What are modules? Which module is used for system calls?
A: Modules are essentially Python files and usually consist of a collection of functions. To show all attributes and methods of a ⟨module⟩ one can use the command dir(⟨module⟩).

There are two modules that can be used [Pan22] for system calls to execute a ⟨command⟩, os and subprocess:

- os.system('⟨command⟩:)
- subprocess.call('⟨command⟩', shell=True)

These calls only execute the ⟨command⟩, but storing their output in a variable gives only the integer 0, denoting a successful execution. To obtain the output produced as string, the call

subprocess.check_output(⟨command⟩, shell=True, text=True)

is needed. Here, the text flag converts the default byte format into a string. Storing its output in a variable is the output of the command, e.g. text printed in bash.

Q: How does file I/O work in Python?
A: For file I/O the following commands are used [Lot23]:

- Open, read/write and close file:

```
fname = open(⟨filename⟩, ⟨file mode⟩)
fname.read()       # read file content
fname.readline()   # read file content line by line
fname.write(⟨data⟩) # write ⟨data⟩ to file
fname.close()
```

The most relevant ⟨file modes⟩ are `'r'`,`'w'`,`'a'` and `'x'` (read only, write only, append, create new file). Binary data can be read, written or appended to using `'rb'`, `'wb'` and `'ab'`. Non-binary and binary read and write can be extended to update the file, i.e. overwrite its content, with `'r+'`, `'w+'`, `'rb+'` and `'wb+'`, while `'a+'` and `'ab+'` are used to read and append.

- Open file such that it is automatically closed afterwards:

```
with open(⟨filename⟩, ⟨file mode⟩) as fname:
    for line in fname:
        ⟨indented block reading/writing fname⟩
```

In practice use the following construct for reading:

```
with open(⟨filename⟩, 'r') as fname:
    for line in fname:
        ⟨indented block using line⟩
```

For reading CSV files, use the csv module:

```
import csv

with open(⟨filename⟩) as fname:
    reader = csv.reader(fname, delimeter=',')
    for line in reader:
        ⟨indented block using line⟩
```

For reading JSON files, use the json module:

```
import json

with open(⟨filename⟩) as fname:
    content = json.load(fname)
    for key,val in content:
        ⟨indented block using key, val in content⟩
```

Q: What are regular expressions (regex) and when are they used?
A: Regular expressions (regex) are sequences of characters that specifies a match pattern in text. They are used to search substrings and operate with them (e.g. save finding, save index of finding, replace finding). Python has the re module to use regular expressions. The most commonly used methods in this module are:

- re.compile(⟨pattern⟩): Compile ⟨pattern⟩ into a Pattern object which can be used for matching using methods of the re module.
- re.search(⟨pattern⟩, ⟨string⟩): Find first location in ⟨string⟩ where ⟨pattern⟩ produces a match and return Match object. Similar methods are re.match and re.fullmatch with the same arguments. While re.match looks for a match only at the beginning of a ⟨string⟩, re.fullmatch checks if the entire ⟨string⟩ is matched by the ⟨pattern⟩.
- re.split(⟨pattern⟩, ⟨string⟩): Split ⟨string⟩ at occurence of ⟨pattern⟩ and return parts as a list. Capturing groups in the ⟨pattern⟩ matching at the start or end of a string lead to empty strings as first or last elements in the resulting list.
- re.findall(⟨pattern⟩, ⟨string⟩): Find all non-overlapping matches of ⟨pattern⟩ in ⟨string⟩ and return them as list of strings (zero or one capturing group) or a list of tuples containing strings (more than one capturing group).
- re.finditer(⟨pattern⟩, ⟨string⟩): Find all non-overlapping matches of ⟨pattern⟩ in ⟨string⟩ as Match objects yielded by an iterator.
- re.sub(⟨pattern⟩, ⟨repl⟩, ⟨string⟩): String obtained by replacing leftmost, non-overlapping occurence of ⟨pattern⟩ in ⟨string⟩ with ⟨repl⟩. ⟨repl⟩ can be a string or a function. Backslash escapes (e.g. `\n`, `\r`) are expanded. The variant re.subn with the same arguments as re.sub does the same, but returns a tuple (new_string, number_of_subs_made).
- re.escape(⟨pattern⟩): Escapes special characters in ⟨pattern⟩. Match an arbitrary literal string containing regular expression metacharacters.

In all methods but re.escape one can specify flags. The most commonly used ones are:

- re.I: Ignore case.
- re.M: Have ^ match the beginning of a string and the beginning of a line (immediately following each newline) and $ match the end of a string and the end of a line (immediately preceding each newline).
- re.S: Have '.' match any character, incluing a newline.

Compared to the bare methods of the re module, Pattern objects can confine their matching for the methods search, match, fullmatch, findall and finditer to the index range [⟨pos⟩, ⟨endpos⟩]. Besides this, the Pattern object has the following attributes:

- flags: Regex flags given to compile() and inside pattern.
- groups: Number of capturing groups in pattern.
- groupindex: Dictionary mapping symbolic group names defined by (?P<id>) to group numbers.
- pattern: Pattern string used to compile Pattern object.

Match objects are returned by several methods and contain True (match) or None (no match). They have the following methods:

- expand(⟨template⟩): Use \⟨number⟩ to expand a match from by a group in a template string.
- group(⟨group IDs⟩): Subgroup matches returned as a single string (one group), a tuple of strings (comma separated list of multiple groups) or the whole match (no argument). Note that group IDs can be indices or names. If a group matches multiple times, only the last match can be accessed.
- groups(): Tuple containing all subgroups of match.
- groupdict(): Dictionary of named subgroups with subgroup name as key and match as value.
- start(group), end(group): Indices of the start and end of the substring matched by the group. If the group did not contribute to a match, both default to −1.
- span(group): Tuple (Match.start(group),Match.end(group))
- pos, endpos: Value of pos, endpos passed to match() or search() giving the Match object.
- lastindex: Integer index of last matched capturing group or None.
- lastgroup: Name of last matched capturing group or None.
- re: Pattern object whose match() or search() gave the Match object
- string: String passed to match() or search() giving the Match object. The substring matched by a capturing group is given by Match.string[Match.start(group):Match.end(group)].

After describing the methods available, the special characters used therein are:

Character	Purpose
.	Match any character except a newline.
^	Match start of string.
$	Match end of string or just before newline at end of string.
∗	Match zero or more repetitions of preceding regex, but as many repetitions as are possible (greedy).
+	Match one or more repetitions of preceding regex, but as many repetitions as are possible (greedy).
?	Match zero or one repetitions of the receding regex (greedy).
∗?, +?, ??	Uses the quantifiers ∗, + or ? in a minimal fashion, i.e. such that they match as few characters as possible (non-greedy).
∗+, ++, ?+	Uses the quantifiers ∗, + or ? without backtracking (possessive), e.g. while the string 'aaaa' is matched by a∗a (a∗ matches three a after backtracking), it is not matched by a∗+a (a∗+ matches four a).
{m}	m copies of the previous regex should be matched.
{m,n}	Between m and n repetitions of the previous regex should be matched, attempting to match as many repetitions as possible (greedy). Omitting m sets the lower limit to zero and omitting n set the upper limit to infinity.
{m,n}?	Between m and n repetitions of the previous regex should be matched, attempting to match as few repetitions as possible (non-greedy).
{m,n}+	Between m and n repetitions of the previous regex should be matched, attempting to match as many repetitions as possible (possessive).

\	Used to escapes special characters or signal special sequence.
[]	Set of characters. Can involve distinct characters (e.g. [amk] matches a or m or k), ranges (e.g. [a-z], [A-Z], [0-9]) and character classes (e.g. [\w]). To match the complement of the set, use ^ as first entry of the set (e.g [^5] matches). Note that the special characters '(', +, ∗, ')', | do not have to be escaped inside a set.
|	Matches either the preceding or trailing regex (e.g. A|B matches A or B). Can be used inside groups. The preceding regex is scanned first and if it matches, the trailing regex is not checked (non-greedy).
(⟨regex⟩)	Matches ⟨regex⟩ inside parentheses, and indicates start and end of a group. The contents of a group can be retrieved after a match and can be used later in the string with the special sequence \⟨number⟩.
(?⟨regex⟩)	Extension notation. The character after the '?' determines the meaning and syntax of the construct.
(?ims)	The group matches an empty string while the letters set the flags re.I, re.M, re.S.
(?:⟨regex⟩)	Non-capturing version of parentheses, i.e. match from group cannot be retrieved.
(?ims-ims:)	The letters set or remove the flags re.I, re.M, re.S for the group.
(?>⟨regex⟩)	Atomic group which does not allow backtracking inside it (does not re-examine match) and is non-capturing.
(?P<⟨name⟩>⟨regex⟩)	Like regular parentheses, but the substring matched by the group is accessible via symbolic group name ⟨name⟩. Note that named groups are still numbered.
(?P=⟨name⟩)	Backreference to named group with ⟨name⟩ which matches the text matched by this earlier group.
(?#⟨text⟩)	Comment.
(?=⟨regex⟩)	Positive lookahead assertion: Full pattern only matches if ⟨regex⟩ matches, but the full match does not include what was matched by the parentheses.
(?!⟨regex⟩)	Negative lookahead assertion: Full pattern only matches if ⟨regex⟩ does not match.
(?<=⟨regex⟩)	Positive lookbehind assertion: Full pattern only matches if current position is preceded by match for ⟨regex⟩, but the full match does not include what was matched by the parentheses.
(?<!⟨regex⟩)	Negative lookbehind assertion: Full pattern only matches if current position is no preceded by match for ⟨regex⟩.
(?⟨ID/name⟩ ⟨yes regex⟩|⟨no regex⟩)	Matches with ⟨yes regex⟩ if the group with ⟨ID/name⟩ matches and with ⟨no regex⟩ if the group with ⟨ID/name⟩ does not match.

Special sequences are numbers or letters preceded by a backslash. The most commonly used ones are:

Character	Purpose
\⟨number⟩	Matches contents of the group with the same ⟨number⟩.
\A	Matches only at start of string.
\b	Matches empty string at beginning or end of word. Thus it is the boundary between a \w and \W character.
\B	Matches empty string not at beginning and not at end of word. It is the opposite of \b.
\d	Matches any decimal digit, e.g. [0-9].
\D	Matches any character not a digit.
\s	Matches whitespace characters, e.g. [\t\n\r].
\S	Matches any character not a whitespace character. It is the opposite of \s.
\w	Matches word characters including alphanumeric characters and the underscore, e.g. [a-zA-Z0-0_]

\W	Matches any character not a word character. It is the opposite of \w.
\Z	Matches only at end of string.

Q: What is the purpose of error and exception handling?
A: While "error" here stands short for "syntax error", an expception is a type of error that occurs when a syntactically correct Python code raises an error. Error and exception handling aim to prevent running code to terminate once an error or exception is encountered.

In Python, exceptions are handled using try-except statements. A full block in exception handling has the following structure [Kos22]:

```
try:
    ⟨Code which can throw an exception⟩
except ⟨Exception kind 1⟩:
    print(⟨Error message clarifying Exception kind 1⟩)
    ⟨Code to recover from Exception kind 1⟩
except ⟨Exception kind 2⟩:
    print(⟨Error message clarifying Exception kind 2⟩)
    ⟨Code to recover from Exception kind 2⟩
...
except ⟨Exception kind n⟩:
    print(⟨Error message clarifying Exception kind n⟩)
    ⟨Code to recover from Exception kind n⟩
else:
    ⟨Code to run when no exception occured⟩
finally:
    ⟨Code running independent of exception occurring⟩
```

The most common types of exceptions are:

- NameError: Name does not exist among local or global variables.
- TypeError: Operation run on inapplicable data type (e.g. int+str).
- ValueError: Operation or function takes in an invalid value of an argument.
- IndexError: Index does not exist in iterable.
- IndentationError: Indentation incorrect.
- ZeroDivisionError: Attempt to divide number by zero.
- ImportError: Import statement incorrect.
- AttributeError: Attempt to assign or refer to an attribute inapplicable for a given Python object.
- KeyError: Key in dictionary is absent.

Futher types of exceptions can be inferred from the Python documentation.

One can also initiate exceptions using the raise keyword and adding an optional output message, e.g. `raise ValueError('Error message')`.

Q: What is the purpose of object oriented programming?
A: Object oriented programming (OOP) is a programming paradigm which serves to organize and structure code. It groups together data (attributes) with the functions (methods) operating on that data. This modularizes the code and isolates different parts of the program from each other. It has the following features:

- Variables: Store data of different types.
- Procedures: Receive input, produce output and manipulate data.
- Classes and objects: Classes serve as blueprints determining the attributes and methods of objectes. Objects are instances of classes which possess actual
- Data abstraction:
- Encapsulation:
- Composition, Inheritance, delegation:
- Polymorphism:
-
-

TODO

Q: What is the purpose of a constructor?
A: A constructor is a method called to create and initialize an object. Its counterpart is a destructor. In Python it is called __init__ and has the method parameters self and initial values for instance attributes. One can use the dataclass decorator available in the dataclasses module to automatically add an __init__ function. Note that this requires a special syntax for the parameters of __init__.

- Q: What are attributes?
  A: Attributes are the data possessed by an object. They are split into class attributes (belong to class, one copy shared by all objects) and instance attributes (belong to individual objects). In Python, class attributes are global variables within the class and can be called as `class.cls_attr`, while instance attributes are called as `self.obj_attr`.

  To indicate that an attribute should only be used within the class, it is convention to precede it by one underscore, e.g. `_secret`. Note that this does not ensure that this attribute is private, i.e. it can still be called, e.g. `obj._secret`. A more active way to prevent accessing an attribute intended to be private consists in using double underscores. This leads to name mangling, i.e. an underscore and the classname are prepended to the attribute name. To call the attribute from outside one would have to use `obj._class__secret`.
- Q: What are methods?
  A: Methods are the functions operating on the attributes. They are split into class methods (belong to class, have only access to class attributes and input parameters) and instance methods (belong to individual objects, have access to instance attributes, class attributes and input parameters). In Python methods are functions defined in the class body.

  Class methods (method belonging to a class), static methods (methods not using attributes of a class but are grouped to a class; subclasses cannot alter them) and abstract methods (methods of a class without implementation intended to be overwritten by the subclasses inhereting them) can be realized in Python using decorators. Abstract methods are available from the abc (abstract base class) module. Class methods can be called as `class.cls_method`, while instance methods are called via `obj.obj_method`.
- Q: What is an iterator?
  A: For an array of integers, the stepsize to get from one entry to the next is simply given by the size of an integer. In case of lists containing more complex data or different types of data, the length of a step is not that simple. An iterator is an object which decouples the iteration algorithm from the data structure and thus allows to iterate over collections of data by providing the correct stepsize. It also allows to iterate over all elements of an unordered structure like a set. If a data collection has an iterator, one can iterate over the entries of the data collection.

1.2 Miscellaneous

- One-line if-else: ⟨task⟩ if (⟨condition⟩) else ⟨alternative⟩
-
-

1.3 Documentation of Task 1

1.3.1 Features

- ReadAble logfiles: See critical errors or warnings with minimal searching
- ExtendAble whitelist: Create your own filter to tweak what your logs contain
- AdjustAble schedule: Create your own schedule by setting up your own cronjob

1.3.2 How to use

There is no need to install additional dependencies since the script only uses modules from the Python standard library [Unk23d] and common Linux commands. It runs without command line arguments. Upon start, it announces the ensuing creation of a cronjob and an example which can be copied by marking it and pressing Ctrl+Shift+c.

```
[jan@linux ~]$ python3 readable_logs.py

| ReadAble Logs |

Create a cronjob to execute this script daily on working days.
Example:
```

```
0  0  *  *  1-5   /usr/bin/python3  ⟨location of script⟩
Press Enter to create a cronjob...
```

Pressing the Enter key displays the crontab file where a cronjob can be set up or the copied example can be entered with Ctrl+Shift+v. After this, the script is executed periodically as determined by the cronjob and the log files (system log file, simplified log file labeled by the date) can be found in the home-directory.

Running the script as root or with root privilege creates the cronjob in the global crontab file and the log files are written to /home.

1.3.3 Short summary

This script produces readable log files which are updated on a periodical basis, either in the calling users home-directory, or in /home by running it as root or with root privilege. The system log file (e.g. Ubuntu: /var/log/syslog) is copied to the calling users home-directory or /home for root. From there, it is read to memory and the simplified content is written to a file labeled by the current date. The original content of the system log file is reduced to those lines containing one of the severity keywords (EMERGENCY, ALERT, CRITICAL, ERROR, WARNING, NOTICE, INFORMATIONAL, DEBUG) in the format " ⟨keyword⟩:" followed by an error message. Furthermore, all content besides the timestamp and the error message are abbreviated. To generate the simplified log files on a regular basis, the user gets the chance to create a cronjob.

1.3.4 Tasks and design choices

After printing the name of the script, a `CronJob` object is created. Its arguments correspond to the different fields in the crontab file. While the path to python3 is resolved in the constructor, the script name is contained in the variable `__file__`. The constructor also checks if the cronjob already exists in the crontab file and sets the `active` attribute accordingly. In case the cronjob is not set up yet, it is added to the crontab file by means of the `add_cronjob` method.

Next, a `LogFile` object is created with the `name` of the file and its `location`. Since the location where the system log file and the modified log files shall be stored are owned by the user, a `User` object with attribute `home` is created. Then the system log file is copied to the users home-directory by means of the `copy_log` method which changes the location of the log file. Finally, the system log file is trimmed in the `trim_log` method.

```
print("| ReadAble Logs |\n")

cronjob = CronJob("0","0","*","*","1-5","python3",f"{__file__}")
if (not cronjob.active):
   cronjob.add_cronjob()

log_file = LogFile("syslog","/var/log")
user = User()
log_file.copy_log(user.home)
log_file.trim_log()
```

▸ Task: Check if the cronjob already exists in the crontab file.

- Implementation: To check if the cronjob already exists in the crontab file, `crontab -l` is executed. In case no cronjob exists, the request returns a failure which raises an CalledProcessError exception [Unk23c]. This is handled in a try-except statement.

```
def __init__(self,m,h,dom,mon,dow,exe,file):
   """Constructs all necessary attributes for the CronJob object."""

   self.__m = m
   self.__h = h
   self.__dom = dom
   self.__mon = mon
   self.__dow = dow
   cmd = f"which {exe}"
   self.__exe = subprocess.check_output(cmd,shell=True,text=True)
   self.__file = file

   self.active = True
   cmd = "crontab -l"
   try:
```

```
16          crontab = subprocess.check_output(cmd,shell=True,text=True, \
17                                            stderr=subprocess.DEVNULL)
18      except subprocess.CalledProcessError:
19          # if no crontab for user
20          crontab = ""
21
22      # check if cronjob for __file__ is set
23      if (self.__file not in crontab):
24          self.active = False
```

Task: Run the script daily at 00:00 o'clock from Mon-Fri.

- Implementation: Inform the user about the ensuing creation of a cronjob, provide an example cronjob and wait for an input to allow the user to copy the example. Then display the crontab file of the user.

```
1  def add_cronjob(self):
2      """Adds cronjob to crontab."""
3
4      print("Create␣a␣cronjob␣to␣execute␣this␣script␣daily␣on␣working␣days.")
5      print(f"Example:\n"+
6            f"{self.__m}\t{self.__h}\t{self.__dom}\t{self.__mon}\t{self.__dow}\t"+
7            f"{self.__exe}␣{self.__file}")
8      input("Press␣Enter␣to␣create␣a␣cronjob...␣")
9
10     # home_path may require root privilege
11     cmd = "crontab␣-e"
12     subprocess.call(cmd,shell=True)
```

- Q: Should I create a cronjob or is there an alternative?
  A: Since the script is supposed to run on Linux servers, using a cronjob is straightforward.
- Q: Should I automatically append the cronjob to the crontab file or let the user enter it?
  A: To automatically append the cronjob to a crontab file manually requires root privilege. The global crontab file in /etc requires root privilege to modify. While the user crontab files can be modified by the user, they are hidden in /var/spool/cron/crontabs, which requires root privilege to access. There exists the crontab module in Python **??**, however it is not part of the Python standard library [Unk23d] and would introduce additional dependencies.

  Hence instead of automatically adding a cronjob, the user is presented with a sample cronjob and the option to add it to the crontab. This allows the user to adjust the scheduling dynamically instead of having to alter the source code.

Task: Copy the system log file into the calling users home-directory.

- Implementation: Using a system call, the system log file is copied from its location to the destination. The location is then updated with the destination. root uses /home instead of their home-directory which is implemented in the constructor of the User class.

```
1  def copy_log(self,destination):
2      """
3      Copies Log to destination.
4
5          Parameters:
6              destination (str): Absolute path
7      """
8
9      cmd = f"cp␣{self.__location}/{self.__name}␣{destination}"
10     subprocess.call(cmd,shell=True)
11     self.__location = destination
12
13 ...
14
15 def __init__(self):
16     """Constructs all necessary attributes for the User object."""
17
18     cmd = "whoami"
19     self.__name = subprocess.check_output(cmd,shell=True,text=True).strip()
20
21     if (self.__name == "root"):
22         # root user: logs in accessible directory
23         self.home = "/home"
24     else:
```

```
25          # regular user: logs in own home-directory
26          self.home = f"/home/{self.__name}"
```

- Q: Are there any obstacles when copying a file from a directory if both are owned by root?
  A: There are no obstacles since the other users have reading rights.
- Q: Are there any obstacles when copying a file into a users home-directory or /home?
  A: There are no obstacles when a user copies a file into their home-directory. To copy files into /home, read them or write into /home requires root privilege.
- Q: Is it more sensible to use multiple system calls or to externalize commands into a bash script and invoke it by a system call?
  A: In the present case, the commands are short enough to keep them as system calls.
- Q: Why does the root user operate in /home and not in its own "home-directory" /root?
  A: The script introduces redundancies and information silos when executed by separate users: Each user has a copy of the same, possibly large, system log file and their own set of filters. Running the script with the central role of root using /root as home-directory does not combat these deficiencies. Therefore the behavior when root runs the script is modified to allow for centralized log files. The code can be further extended to modify the permissions of users to access and modify the entries of the log files.

▸ Task: Read the system log file.

- Implementation: Read the system log file by means of a generator, one line at a time.

```
1  def __read_log(self):
2      """Reads from Log file, one line at a time."""
3
4      fname = f"{self.__location}/{self.__name}"
5      with open(fname,"r") as logfile:
6          for line in logfile:
7              yield line
```

- Q: Does the format of the system log file require any specific handling (e.g. unpacking, decompression, file conversion)?
  A: The system log file is a text file which does not require specific handling.
- Q: Does the system log file fit into memory when reading it?
  A: This is unclear. A generator is used to read the system log file, one line at a time.
  The current implementation could be modified to read the file in batches. This alone would not significantly alter the performance without further modifications for the following reasons:
  - The generator does not close the file between consecutive reads. Reading one line at a time or batches via a generator opens and closes the file only once.
  - The filter operates on a single line, hence to improve performance when using batchwise reads would require to modify application of the filter. One option would be to move the filter into a separate function operating on a list which can then be executed concurrently using the threading module.

  To keep the code simple, one line at a time is read.
- Q: Would it be expedient to store the content of the system log file in a specific data structure?
  A: Since the size of the system log file is unclear, it is read line by line. With the size of the system log file unclear, it is also unclear how many lines matching the whitelist there are. Appending the lines approved by the whitelist to a list is therefore not sensible.
  Since the overall task is to filter the file and not to display it, modifying the representation of the data by means of the `__repr__` and `__str__` method are not necessary.
  Each line is kept as a string to use regular expressions for filtering.

▸ Task: Modify log entries, i.e. what to filter for and how to filter.

- Implementation: Set up a whitelist with expressions to search for. Start the generator [Unk23a] used to write the modified logs to file. While looping through the system log file, each line undergoes the following examination:
  - Check if an entry of the whitelist is contained in the line
    ↪ No entry of the whitelist is contained in the line: Skip to the next line
    ↪ An entry of the whitelist is contained in the line: Obtain the timestamp and the error message behind the whitelist entry, combine both into one strng separated by a hyphen, write the modified line to file

After the entire system log file is handled, close the write generator.

```
def trim_log(self):
    """
    Trims Log to improve readability.

    Modify whitelist to control filter.
    """

    # whitelist of filter
    whitelist = ["␣EMERGENCY:","␣ALERT:","␣CRITICAL:","␣ERROR:",\
                 "␣WARNING:","␣NOTICE:","␣INFORMATIONAL:","␣DEBUG:"]

    write_gen = self.__write_log()
    write_gen.send(None)

    for line_log in self.__read_log():
        # check if line contains whitelist entry
        is_allowed = [(wl_entry in line_log) for wl_entry in whitelist]
        if (any(is_allowed)):
              wl_entry = whitelist[is_allowed.index(True)]
        else:
              # omit lines without whitelist entry
              continue

        # keep timestamp, messages with whitelist entry
        time_pattern = r"^(.+?)\s"
        wl_pattern = r"("+wl_entry+r".+?)$"
        timestamp = re.search(time_pattern,line_log).group(1)
        wl_message = re.search(wl_pattern,line_log).group(1).strip()
        line_mod = "␣-␣".join([timestamp,wl_message])

        write_gen.send(line_mod)

    # close generator
    write_gen.close()
```

- Q: Would it be expedient to convert the content of the system log file into a specific data structure to improve filtering?
  A: Each line is kept as a string to use regular expressions for filtering. Splitting each line into a list of words to filter it word by word would introduce two obstacles: Filtering for anything but words requires additional effort and the need to parse each list would harm the performance.
- Q: Which method should be used to filter the file: regular expressions, string methods or manual filtering (e.g. sliding window)?
  A: To filter efficiently for arbitrary patterns, regular expressions should be used. They are applied to each line of the system log file separately.
- Q: Should the original data be overwritten or should a new data structure with the modified data be generated?
  A: Since strings are immutable in Python, in place modifications are not possible.

▸ Task: Write the modified data of the system log file into a file within the home-directory labeled by the current date.

- Implementation: Get the current date using the date method of the datetime module. Write the data to a file named after the current date using a generator, one line at a time.

```
def __write_log(self):
    """Writes trimmed Log to file YYYY-MM-dd, one line at a time."""

    fname = f"{self.__location}/"+date.today().strftime("%Y-%m-%d")
    with open(fname,"w") as logfile:
        while True:
              data = (yield)
              logfile.write(data+"\n")
```

- Q: Are there any obstacles when writing into a file within a users home-directory or /home?
  A: There are no obstacles when a user writes into a file within their home-directory. To write into files within /home requires root privilege.
- Q: Why is a generator used to write out the data?
  A: Since it is unknown whether the system log file fits into memory, it is also unknown whether

the lines matched by the whitelist would fit into memory. Storing them in a list is thus not sensible. Since the data to be written emerges one line at a time and a generator does not introduce additional openings and closings of files, a generator is used.

Overall, reading and writing one line at a time distributes the file I/O evenly over time and avoids bottlenecks where data has to be read or written to continue execution. Reading and writing in batches is the next logical step. However to improve performance, additional concurrent execution by means of the threading module needs to be added.

- Q: How do I get the system date into Python? Is there a module for that purpose or do I have to perform a system call to obtain it from the Linux date command?
  A: The date command in the module datetime provides the current date in Python.

1 3 5 Planned features

- Whitelist: Since the severities are not by default contained in the system log file **??** under ubuntu, the rsyslog templates have to be adapted or introduced to add them to the error messages. This should not be done automatically since it could break other applications, but the documentation should point this out and add a section how to add a template.
- Concurrency: Read and write the log file in batches and apply the filter concurrently.
- Access control: Add a group field to the `User` class to set which users have access to the modified log files.
- Presentation: Add an option to display the logs to allow more highlighting.
- Command line: Add command line arguments for the user to filter the log file similar to journalctl.

2 Meet SQL

2 1 Questions

2 1 1 Relational databases (RDBs)

- Q: How is a database organized?
  A: A database consists of an organized collection of structured information, typically in the form of tables with columns and rows.
- Q: What is a RDB?
  A: A RDB is a type of database that stores and provides access to data points that are related to one another.
- Q: What is a database management system (DBMS)?
  A: A DBMS is software for storing and retrieving data while considering appropriate security measures. Given a request for data from an application, it manipulates the database to provide the data.
- Q: What are the advantages of database management systems (DMBSs) compared to file systems?
  A: An incomplete list of advantages of DBMSs over file systems are the following:
  - variety of techniques to store and retrieve data efficiently
  - uniform procedures to administer data
  - application programmer never exposed to details of data representation and storage
  - offers data integrity and security
  - reduced development time for applications
  - requires less memory
  - reduction of redundancy
  - data independence

  While there are advantages of databases over file systems relevant for specific applications, file systems also have their benefits.
- Q: What is meant when discussing the redundancy of data?
  A: Redundancy of data refers to having multiple copies of the same data within a database. While redundant data requires more storage space, it can improve performance since queries can be less complex and require less operations and thus time to complete. Another relevant term is data integrity, which refers to the maintenance and assurance of data accuracy and consistency.
- Q: What are relations in a database?
  A: The term relations refers to the tables forming the RDB, i.e. one relation is one table.
- Q: What are tuples in a database?
  A: The term tuples refers to the rows in the tables forming the RDB, i.e. one tuple is one row in a table. Using the terminology, tuples are the rows in the relations.
- Q: What are attributes in a database?
  A: The term attributes refers to the columns in the tables forming the RDB, i.e. one attribute is one column in a table. Using the terminology, attributes are the columns in the relations.
- Q: What are primary keys in a database?
  A: The term primary keys refers to the columns of the tables which act as unique identifiers of the rows in the tables, i.e. one primary key is the column of a table which uniquely identifies the row. Using the terminology, primary keys are the attributes of the relations which uniquely identify the tuples of the relations.

  Other types of keys are alternate keys and foreign keys. Alternate keys are alternative unique identifiers besides the primary keys which often consist of several columns. When adding a private key of one relation to another relation, it is called a foreign key.
- Q: What set of properties do database transactions have to fulfill and why?
  A: In order to guarantee data validity despite errors, power failures or other mishaps, a sequence of database operations has to satisfy the ACID properties:

- Atomicity: Each transaction is treated as a single unit which either succeeds completely or fails completely. If any of the statements constituting the transaction fails, the entire transaction fails and leaves the database unchanged. This prevents partial updates to the database. Hence the transaction also cannot be observed to be in progress, it either has not occured yet or it is already finished.
- Consistency: Any transaction can only bring the database from one consistent state to another while preserving database invariants. This means any data written to the database must be valid according to all defined rules, including constraints, cascades (cascading rollbacks return a database after a transaction failure back to a previous state), triggers (automatic responses to certain events) and any combination thereof. This prevents database corruption by illegal transactions. Referential integrity (for all references from one attribute of a relation to another attribute, either of the same or a different relation, a value must exist) guarantees the primary key-foreign key relationships.
- Isolation: Transactions occuring concurrently leave the database in a state that would have been obtained if the transactions were executed sequentially. This ensures that the transactions are not performed in an arbitrary order.
- Durability: Guarantees that once a transaction has been committed, it will remain committed, even in case of system failure.

Q: What is database normalization?
A: Database normalization describes the process of structuring a relational database following a series of normal forms to reduce data redundancy and improve data integrity. Thereby attributes and relations of a database are organized such that their dependencies are enforced by database integrity constraints. To normalize a database either a new database design is created (synthesis) or an existing database design is improved (decomposition).

Q: What types of integrity constraints are there?
A: There are three types of integrity constraints:

- Entity integrity: Every relation must have a primary key which must be unique and not null.
- Referential integrity: Any foreign key can refer either to the primary key value of some table in the database or to a null value. The null value can mean either that there is no relationship or that the relationship is unknown.
- Domain integrity: All attributes in a RDB must be declare upon a defined domain of non-decomposable/atomic data items which are of the same type.
- User-defined integrity: User-specified rules.

Q: Which types of anomalies can arise in relations that have not been sufficiently normalized?
A: There are three types of anomalies that can arise when relations have not been sufficiently normalized:

- Insertion anomaly: If a newly inserted tuple has an attribute value which is undefined, an insertion anomaly occurs. In ensuing queries this tuple will be omitted when requesting the undefined attribute.
- Update anomaly: In case the same information is contained in multiple tuples, updates to only some of them lead to an update anomaly. In ensuing queries the relation will provide conflicting answers.
- Deletion anomaly: If the value of an attribute in a tuple is deleted and thus undefined, an deletion anomaly occurs. In ensuing queries this tuple will be omitted when requesting the undefined attribute.

Q: What constraints are enforced in each normal form (NF) considering UNF, 1NF, 2NF and 3NF? How are lower NFs decomposed into the next higher NF?
A: The first three NFs require the following constraints and decompose into higher NFS as follows:

- Unnormalized form: There are no duplicate tuples.
- First normal form (1NF):
  - Requirement: Relation must be of UNF.
  - Constraint: No attribute contains relations.
  - Decompose into 1NF: Move relations out of attributes into separate relations with the original primary key as foreign key.
- Second normal form (2NF):
  - Requirement: Relation must be of 1NF.
  - Constraint: In case of a composite primary key, each attribute which is not part of the primary key of the relation must depend on the complete primary key and not only on a part of it.

- Decompose into 2NF: Move attributes which depend only on a part of the primary key into other relations and use the part of the primary key they depend on as the complete primary keys of the new relations.

- Third normal form (3NF):
  - Requirement: Relation must be of 2NF.
  - Constraint: Attributes must only depend on the primary key directly. Transitive dependence on the primary key, i.e. an attribute attr1 depending on an attribute attr2 which in turn depends on the primary key is not allowed.
  - Decompose into 3NF: Move attributes which depend only transitively on the primary key into other relations and use the attributes through which they depended on the primary key as new primary keys in these new relations.

A colloquial summary of the first three normal forms is: "Every non-key attribute must provide facts about the key (1NF), the whole key (2NF) and nothing but the key (3NF), so help me Codd."

2.1.2 Entity relation model (ERM)

Q: What are entities, relationships and attributes in an ERM?
A: An entity is an item which exists independently of other items, can be uniquely identified and possesses attributes. It can exist physically (e.g. an object) or logically (e.g. a concept). Entities can be considered instances of entity-types, while an entity-type is a category. Put simple, an entity is given by a proper noun, while an entity-type is given by a common noun.

A relationship captures how entities are related to one another. They describe the participation of multiple entities in a relationship, contain attributes, indicate dependency and exclusivity constraints and create hierarchies. Put simple, a relationship is given by a transitive verb.

Both, entities and relationships can have attributes which are properties providing descriptive information about them. There are two types of attributes, identifiers and descriptors. Put simple, the attribute of an entity is given by an intransitive verb or an adjective and the attribute of a relationship is given by an intransitive verb or an adverb.

Q: What types of relationships are there?
A: Relationships can be distinguished based on different qualifiers.

Based on the number of different types of entity sets involved in a relationship set (degree of relationship set), there are three types of relationships:

- Unary relationships: Only one type of entity sets participates in a relationship set. Example: The relationship between an employee and a manager involves only the Employees entity set since both are employees of the sam company.
- Binary relationships: Two types of entity sets participate in a relationship set. Example: The relationship between a customer and a product involves the entity sets Customers and Products.
- n-ary relationships: n types of entity sets participate in a relationship set. Example: The relationship between a teacher giving a class with a student involves the entity sets Teachers, Classes and Students.

Assume a binary relationship between two entity sets A and B. Based on the number of times an entity of the same set participates in a relationship (cardinality), there are four additional types of relationships:

- One-to-one (1:1): Each entity of A and B participates only once in a relationship. The mapping $f : A \rightarrow B$ is bijective.
- One-to-many (1:m): While the entities of A can participate multiple times in a relationship, the entities of B participate only once. The mapping $f : B \rightarrow A$ is surjective.
- Many-to-one (n:1): While the entities of A participate only once in a relationship, the entities of B can participate multiple times. The mapping $f : A \rightarrow B$ is surjective.
- Many-to-many (n:m): Both, the entities of A and the entities of B can participate multiple times in a relationship.

Relationship sets are characterized by the optionality (minimum cardinality) and maximum frequency (maximum cardinality) determining the range in which the number of allowed relationships lies.

Q: What types of attributes are there?
A: There are four types of attributes:

- Key attribute: Unique identifier of a tuple of an entity set.

- Composite attribute: One attribute composed of other attributes.
- Multivalued attribute: One attribute possessing more than one value.
- Derived attribute: Attribute which is not possessed by the entity itself, but which is derived from other attributes or other entities.

Q: How does one choose a key attribute?
A: The key attribute of an entity must fulfill the following criteria:

- Definitive: Must alway exist.
- Uniqueness: Must be unique.
- Minimal: May be composite, but must consist of the smallest number of attributes necessary.
- Stable: Must not change over time,
- Factless: Must not entail hidden information.
- Accessible: Must be available when data is created.

Q: What is and how does one create an entity relationship diagram (ERD)?
A: An ERD shows how entity sets are connected by relationship sets. It helps to design databases in an efficient way.

There are a variety of notations to draw an ERD [Gcc19]. The most commonly used is the IE or Crow's foot notation.

- Entity:
  - Rectangular box with the entity name ⟨ent name⟩ in the header and a list of attributes below with different specifiers.
  - Key attribute: ⟨attr name⟩ ⟨PI⟩ ⟨M⟩
  - Alternate attribute: ⟨attr name⟩ ⟨AI⟩ ⟨M⟩
  - Mandatory attribute: ⟨attr name⟩ ⟨M⟩
  - Optional attribute: ⟨attr name⟩
-
-
-

TODO

Q: How do the terms in an ERM relate to the terms in RDBs?
A: The expressions used in an ERM map to terms used in RDBs in the following way:

ERM	RDBs	Physical
Entity/Relationship type	Relation	Table
Entity/Relationship	Tuple	Row
Attribute	Attribute	Column
Key attribute	Primary key	Unique column characterizing a row

2.1.3 Structured query language (SQL)

Q: What are the most common SQL management commands on the command line?
A: TODO

Q: How does one create users and control permissions in Postgresql?
A: The commands to create an user and control permissions are:

- Create a new user:
  CREATE USER ⟨username⟩;
- Grant CONNECT to the database (physically separate collections of tables):
  GRANT CONNECT ON DATABASE ⟨db name⟩ TO ⟨username⟩;
- Grant USAGE on schema (logically separate collections of tables):
  GRANT USAGE ON SCHEMA ⟨schema name⟩ TO ⟨username⟩;
- Grant on all tables for data manipulation language (DML) statements (SELECT, INSERT, UPDATE, DELETE):

GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA ⟨schema name⟩ TO ⟨username⟩;

- Grant all privileges on all tables in the schema:
  GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA ⟨schema name⟩ TO ⟨username⟩;
- Grant all privileges on all sequences in the schema:
  GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA ⟨schema name⟩ TO ⟨username⟩;
- Grant all privileges on the database:
  GRANT ALL PRIVILEGES ON DATABASE ⟨db name⟩ TO ⟨username⟩;
- Grant all privileges on schema:
  GRANT ALL PRIVILEGES ON SCHEMA ⟨schema name⟩ TO ⟨username⟩;
- Grant permission to create a database:
  ALTER USER ⟨username⟩ CREATEDB;
- Make a user a superuser:
  ALTER USER ⟨username⟩ WITH SUPERUSER;
- Remove superuser status:
  ALTER USER ⟨username⟩ WITH NOSUPERUSER;
- Grant permissions for newly created tables by altering the default:
  ALTER DEFAULT PRIVILEGES
  FOR USER ⟨username⟩
  IN SCHEMA ⟨schema name⟩
  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO ⟨username⟩;

Q: What are SQL queries?
A: A SQL query corresponds to a request sent to the database to receive data.

Q: What are the most common SQL query commands?
A: The most common SQL query commands are:

Command	Output
SELECT ⟨attr⟩ FROM ⟨rel⟩;	Return the values of the attribute ⟨attr⟩ from the relation ⟨rel⟩.
SELECT ∗ FROM ⟨rel⟩;	Return the values of all attributes from relation ⟨rel⟩.
SELECT DISTINCT ⟨attr⟩ FROM ⟨rel⟩;	Return distinct values of the attribute ⟨attr⟩ from relation ⟨rel⟩.
SELECT ⟨attr⟩ INTO ⟨rel1⟩ FROM ⟨rel0⟩	Copy values of the attribute ⟨attr⟩ from relation ⟨rel0⟩ into relation ⟨rel1⟩.
SELECT TOP ⟨pct⟩ ∗ FROM ⟨rel⟩;	Return the top ⟨pct⟩ percent of tuples from a relation ⟨rel⟩.
SELECT ⟨attr⟩ AS ⟨alias⟩ FROM ⟨rel⟩;	Rename the attribute ⟨attr⟩ of the relation ⟨rel⟩ with the alias ⟨alias⟩.
SELECT ⟨attr⟩ FROM ⟨rel⟩ AS⟨alias⟩;	Return values of the attribute ⟨attr⟩ while renaming the relation ⟨rel⟩ with alias ⟨alias⟩.
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨cond(attr1)⟩;	Return values of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the values of the attribute ⟨attr1⟩ satisfy the condition ⟨cond(attr1)⟩. Note that the condition can also restrict the values of the attribute ⟨attr0⟩ or multiple attributes. The condition involves regular comparison operators like =, !=, >, <, >= or <=.
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨cond0(attr1)⟩ AND ⟨cond1(attr2)⟩;	Return values of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the values of attribute ⟨attr1⟩ satisfy the condition ⟨cond0(attr1)⟩ and the values of attribute ⟨attr2⟩ satisfy the condition ⟨cond1(attr2)⟩. Note that the attributes ⟨attr1⟩ and ⟨attr2⟩ can also be ⟨attr0⟩.

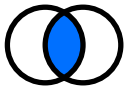
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨cond0(attr1)⟩ OR ⟨cond1(attr2)⟩;	Return values of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the values of attribute ⟨attr1⟩ satisfy the condition ⟨cond0(attr1)⟩ or the values of attribute ⟨attr2⟩ satisfy the condition ⟨cond1(attr2)⟩. Note that the attributes ⟨attr1⟩ and ⟨attr2⟩ can also be ⟨attr0⟩.
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨attr1⟩ BETWEEN ⟨val0⟩ AND ⟨val1⟩;	Return values of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the values of attribute ⟨attr1⟩ lie in the range [⟨val0⟩:⟨val1⟩]. Note that the attribute ⟨attr1⟩ can also be ⟨attr0⟩.
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨attr1⟩ LIKE ⟨pat⟩;	Return values of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the values of attribute ⟨attr1⟩ follow the pattern ⟨pat⟩. Note that the attribute ⟨attr1⟩ can also be ⟨attr0⟩.
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨attr1⟩ IN ⟨lst⟩;	Return the value of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the value of attribute ⟨attr1⟩ is in the list ⟨lst⟩ [syntax: ('A','B','C')]. Note that the attribute ⟨attr1⟩ can also be ⟨attr0⟩.
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨attr1⟩ IS NULL;	Return values of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the values of attribute ⟨attr1⟩ are NULL.
SELECT ⟨attr0⟩ FROM ⟨rel⟩ WHERE ⟨attr1⟩ IS NOT NULL;	Return values of the attribute ⟨attr0⟩ from the relation ⟨rel⟩ where the values of attribute ⟨attr1⟩ are not NULL.
CREATE DATABASE ⟨db name⟩;	Create a new database with name ⟨db name⟩ given the user has the correct admin rights.
CREATE TABLE ⟨tbl name⟩ ();	
CREATE INDEX ⟨idx name⟩ ON ⟨rel⟩	
CREATE VIEW	
DROP DATABASE ⟨db name⟩;	
DROP TABLE ⟨tbl name⟩	
DROP INDEX ⟨idx name⟩	
UPDATE ⟨rel⟩ SET ⟨attr0⟩ = ⟨val0⟩ WHERE ⟨attr1⟩ = ⟨val1⟩	
DELETE FROM ⟨rel⟩ WHERE ⟨attr⟩ = ⟨val⟩	

- ◦ %⟨text⟩: Values begin with ⟨text⟩.
  ◦ %⟨text⟩%: Values contain ⟨text⟩.

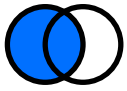
    - ⟨text⟩%: Values end with ⟨text⟩.
    - ⟨text0⟩%⟨text⟩: Values begin with ⟨text0⟩ and end with ⟨text1⟩.
    - _⟨text⟩%: Values which contain ⟨text⟩ starting from the second character.
    - ⟨text⟩_%: Values begin with ⟨text⟩ followed by another character. Additional characters can be considered by adding more underscores.
  - WHERE <attr> IS NULL: Return tuple with NULL value in attr.
  - IS NOT NULL:
  -
  -
  -
  -
  -
  - UPDATE:
  - DELETE:
  - INSERT INTO:
  - CREATE DATABASE:
  - ALTER DATABASE:
  - CREATE TABLE:
  - ALTER TABLE:
  - DROP TABLE:
  - CREATE INDEX:
  - DROP INDEX:
  -
  -
  -
  -

▸ Q: What are JOIN statements and what types of JOINs are there?
A: JOIN statements combine tuples of two or more relations based on the attribute relating them.
There are four types of joins:

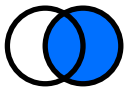
- INNER JOIN: Return tuples which have matching values in the involved relations.



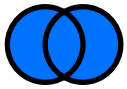
- LEFT (OUTER) JOIN: Return all tuples from the left relation and the matched tuples from the right relation.



- RIGHT (OUTER) JOIN: Return all tuples from the right relation and the matched tuples from the left relation.



- FULL (OUTER) JOIN: Return all tuples when there is a match in either the left or the right relation.



2.1.4 Miscellaneous

▸ In production code SELECT ∗ is considered bad style
▸ It is good style to qualify all column names in a join query ⟶query will not fail when there are duplicate columns
▸ Use explicit joins than performing joins in the WHERE statement
▸ WHERE filters rows before groups and aggregates, HAVING filters after groups and aggregates
▸ WHERE must not contain an aggregate clause, HAVING must always contain an aggregate clause
▸ Can use subqueries

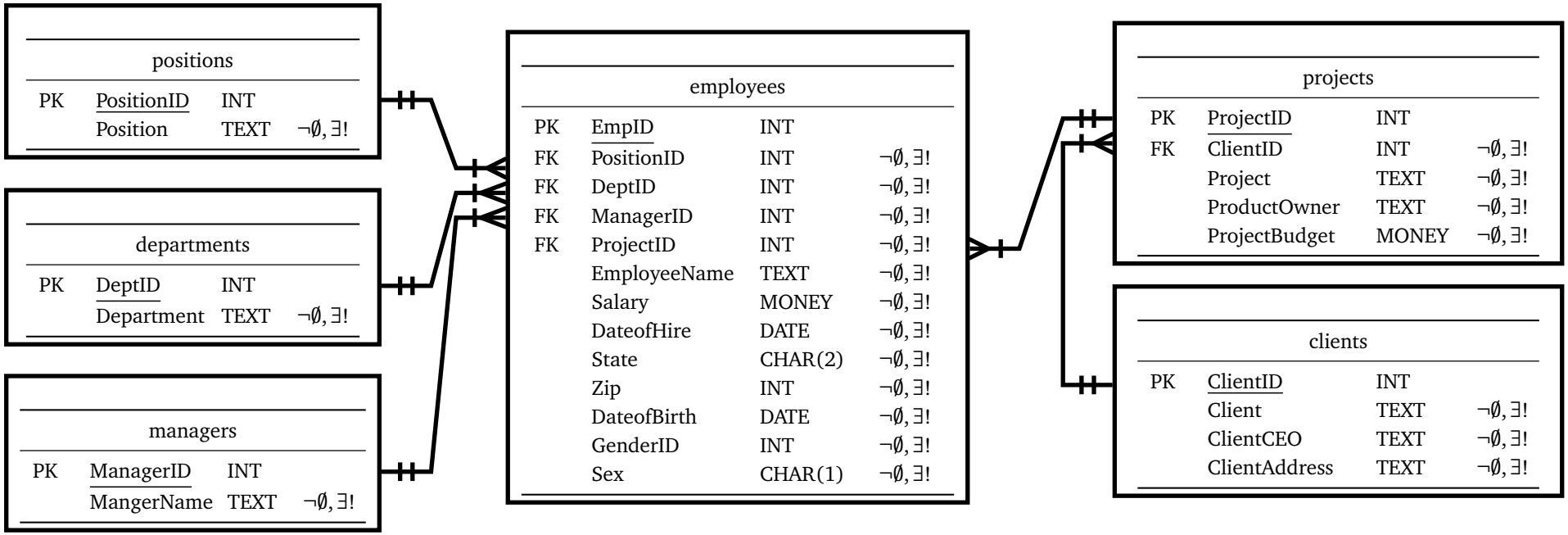
- FILTER similar to WHERE
- Warning: DELETE FROM ⟨table name⟩ without qualification deletes all rows
- Good SQL database design: Make liberal use of views (give name to the result of a query, can be referred to like an ordinary table with DML statements)
- Use foreign keys to maintain referential integrity (prevent table B with a foreign key to get new entries, if the new foreign key value is not yet a primary key value in table A)! Improves quality of database applications
- Comments: –
- Transaction blocks:
  - Used to combine multiple statements into an atomic one
  - Check documentation of interface if the client libraries issue transaction blocks automatically
  - Structure:
    BEGIN;
    DML statements;
    COMMMIT;
  - SAVEPOINT: State of database to which it can revert by means of a ROLLBACK TO statement within a transaction block
  - SAVEPOINTs can be released to free resources, but all SAVEPOINTs defined after the one released or rolled back to are released!
  - SAVEPOINTs are contained in a transaction block →rolled-back actions not visible
  - If a transaction block ends in an aborted state by the system due to an error: ROLLBACK TO only way to regain control
  - Structure:
    BEGIN;
    DML statements;
    SAVEPOINT save1; DML statements;
    ROLLBACK TO save1; DML statements;
    COMMIT;
- Window functions:
  - Performs calculation across several related rows without aggrgating them
  - Structure:
    SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM empsalary;
  - PARTITION BY (optional) gives the attribute by which the output is processed in groups
  - ORDER BY (optional) specifies the order in which the rows are processed in the windows function
  -
- Inheritance:
  - Inherit attributes from one table in another table and links them together, such that a SELECT of a parent table includes the child table. To only select a parent table, SELECT ONLY is used. The ONLY keyword exists also for other DML statements
  - Structure:
    CREATE TABLE ⟨table1⟩ (
    ...
    );

    CREATE TABLE ⟨table2⟩ (
    ...
) INHERITS (⟨table2⟩);

  - Inheritance has not been integrated with unique constraints or foreign keys
- show users: \du
- show databases and permissions: \l
- show schemas: \dn
- show tables: \dt
- switch databases: \c ⟨db name⟩
- show current user: SELECT current_user, session_user;
- change user: SET ROLE ⟨user name⟩

- start a session as another user: psql -d ⟨db name⟩ -U ⟨user name⟩
- close connection to database: \c postgres
-
-
-

2 2 Documentation of Task 2



2.1 – Entity-Relationship diagram (ERD).

2.2.1 Features

- Create-mode:
  - Get an OverVIEW: Create a database and populate it with data.
  - Test-suite: Run a set of SQL-queries to test the database.
- Interactive-mode: Process SQL-queries on the created database interactively.

2.2.2 How to use

```
[jan@linux ~]$ python3 create_db_OOP.py -c

| OverVIEW of AlphaTech Consulting |

Provide login details for the database owner.
Username: ser
Password:

| Create-mode |

Creating the database "alphatech"...
Database created.

Creating and filling the database relations...
Database relations created and filled.

Adding constraints...
Constraints added.

Creating sample queries and output...
Created sample queries and output.
```

```
[jan@linux ~]$ python3 create_db_OOP.py -i

| OverVIEW of AlphaTech Consulting |

Provide login details for the database owner.
Username: ser
Password:

| Interactive-mode |

Enter query:
SELECT * FROM departments;
+------------------------------+
| DeptID | Department          |
|--------+---------------------|
| 4      | Software Engineering |
| 3      | IT/IS               |
| 1      | Admin Offices       |
| 2      | Sales               |
| 0      | Executive Office    |
+------------------------------+

Press q+Enter to quit or Enter to continue... q
```

2.2.3 Dependencies

Python Standard Library:

- sys
- getpass
- csv
- itertools
- textwrap

- functools

Third-party libraries:

- psycopg2
- rich
- PostgreSQL

To install the first two third-party libraries for Python, pip can be used.

```
pip install psycopg2-binary
pip install rich
apt install postgresql
```

2.2.4 Technical summary

2.2.5 Features & Extensibility

2.2.6 Tasks & design choices

- avoid making attr_dict a class attribute, since this associates the class with one file, while hypothetically several files can be opened
- Task: Check if the cronjob already exists in the crontab file.
  -
- Task: Run the script daily at 00:00 o'clock from Mon-Fri.
  -

2.2.7 Planned features

- TODO: psycopg2.errors.UndefinedColumn

Bibliography

[Gcc19] E. Gcc, *Don't get wrong! Explained guide to choosing a database design notation for ERD in a while*, https://medium.com/@ericgcc/dont-get-wrong-explained-guide-to-choosing-a-database-design-notation-for-erd-in-a-while-7747925a7531, Accessed on 09.11.2023, 9:20 am, 2019.

[Kos22] E. Kosourova, *Python Exceptions: The Ultimate Beginner's Guide (with Examples)*, https://www.dataquest.io/blog/python-exceptions/, Accessed on 31.10.2023, 10:20 am, 2022.

[Lot23] M. Lotfinejad, *Input and Output*, https://www.dataquest.io/blog/read-file-python/, Accessed on 31.10.2023, 10:20 am, 2023.

[Pan22] Pankaj, *Python System Command - os.system(), subprocess.call()*, https://www.digitalocean.com/community/tutorials/python-system-command-os-subprocess-call, Accessed on 31.10.2023, 10:20 am, 2022.

[San22] A. Sannikov, *Python Data Structures: Lists, Dictionaries, Sets, Tuples (2023)*, https://www.dataquest.io/blog/data-structures-in-python/, Accessed on 31.10.2023, 10:20 am, 2022.

[Unk23a] Unknown, *6.2.9.1. Generator-iterator methods*, https://docs.python.org/3/reference/expressions.html#generator-iterator-methods, Accessed on 07.11.2023, 8:00 am, 2023.

[Unk23b] Unknown, *Control Structures in Python*, https://www.javatpoint.com/control-structures-in-python, Accessed on 03.11.2023, 1:30 pm, 2023.

[Unk23c] Unknown, *subprocess - Subprocess management*, https://docs.python.org/3/library/subprocess.html, Accessed on 07.11.2023, 9:00 am, 2023.

[Unk23d] Unknown, *The Python Standard Library*, https://docs.python.org/3/library/index.html, Accessed on 06.11.2023, 6:50 pm, 2023.