Informatik I

Introduction to Programming

Assessment Exam Winter 2019

General Guidelines:

- You can reach a maximum of **90 points**, achievable by completing all tasks.
- You have **90 minutes** to complete the test.
- Please check that you have received all 12 pages of this exam.
- Use a **black or blue, permanent pen** for this exam. It is **not allowed** to write with **green or red pens** or with a **pencil**. Affected answers will not be considered in the grading.
- Do not remove the stapling of this test.
- Please write down your **last name** and your **student id** at the bottom of **each** page.
- You may use a **hand-written formulary** (DIN-A5, two-sided) that clearly states your name.
- Non-native speakers may use a **dictionary**.
- You must **not** use any additional resources. If you use any unfair or unauthorized resources or if you copy from a fellow student, you have to hand in your test immediately and it will be considered as failed. Additionally, there will be a disciplinary enquiry.
- Use **Python 3.7** and its corresponding functions for your answers. It is not allowed to use predefined functions if the task description asks you to implement them.
- We have included a list of helpful Python functions on the last page.
- You are not allowed to change predefined method signatures or variable names in the exam.
- **You acknowledge the following points by returning your exam:**
  - I have read and understood these guidelines.
  - I am mentally and physically fit to solve the exam.
  - The room is adequate and I can work on the exam undisturbed.
- Any disturbance during the exam has to be reported to the supervisory staff immediately.

Additional Notes for the English Exam

- This English version of the exam is a translation service for the students.
- If differences exist between the two translations, the German version is decisive.
- You can use English language in your textual answers.
- Provide your answers in the German exam, **answers in this English version will be ignored**.

Task 1: General Questions 20 Points

This task lists several small Python snippets, each of which has an expression in the last line. Write down the *type* and the *value* of these expressions. Remember that also expression without *explicit* values do have an *implicit* type and value. In case the provided snippet crashes with an error, state *Exception* as type and *error* as value. If running a snippet results in an endless loop, state *NoneType* as type and *endless loop* as value.

Note: Naming the simple type is enough, e.g., *int* or *integer*, you can omit the module.

Note: The snippets are to be considered separately. They do not have side effects on each other.

a) 2 Points

```
not ()
```

Type: Value:

b) 2 Points

```
print("Hello World!")
```

Type: Value:

c) 2 Points

```
input("How tall are you?") # assume that the user enters 1.83
```

Type: Value:

d) 2 Points

```
l = [1, 2, 3, 4]
l[1:3] = []
l
```

Type: Value:

e) 2 Points

```
def fun(l):
   if len(l) > 0:
      return fun(l[0])
   else:
      return 42
l = []
l.append(l)
fun(l)
```

Type: Value:

f) 2 Points

```
class Person:
   def get_name(self):
      return self.name
p1 = Person("Adam")
p2 = Person("Bran")
p1.get_name()
```

Type: Value:

g) 2 Points

```
class X: pass
class Y(X): pass
1 if isinstance(Y(), object) else 2.3
```

Type: Value:

h) 2 Points

```
a = 2
b = 3.0
assert a < b
a*b
```

Type: Value:

i) 2 Points

```
try:
   x = None
   raise IndexError()
   x = 1
except IndexError:
   x = 2.0
except:
   x = True
else:
   x = -1+0j
finally:
   x = "fin"
x
```

Type: Value:

j) 2 Points

```
try:
   x = 42 / 0
finally:
   x = 1
x
```

Type: Value:

Task 2: Hailstone Sequence 10 Points

Write a function that, starting from an arbitrary *positive* integer n, generates a list of integers. The list should start with n, followed by a sequence, which is generated according to two rules:

- if the current element is even, divide it by 2 to generate the next element
- if the current element is odd, multiply it by 3 and add 1 to generate the next element

End the sequence once you reach a value of 1 to prevent an endless continuation (1, 4, 2, 1, ...). This resulting sequence is called the *hailstone sequence*.

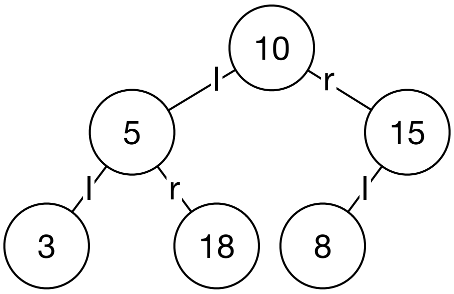
```
def hailstone(n):
```

```
assert hailstone(1) == [1]
assert hailstone(3) == [3, 10, 5, 16, 8, 4, 2, 1]
assert hailstone(7) == [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, ...]
```

Task 3: Recursion 10 Points

A binary tree is a data structure in which each `Node` has a value and two (optional) children, which are referred to as the left and the right child. `root` is illustrated in the image next to the code.

```
class Node:
   def __init__(self, v, l=None, r=None):
      self.v = v
      self.l = l
      self.r = r
root = Node(10, \
  Node(5, Node(3), Node(18)), \
  Node(15, Node(8)))
```



Implement the function `range_sum` that, given a binary tree and the two boundaries `lower` and `upper`, returns the sum of all values `v` contained in the tree, for which `lower <= v < upper`.

```
def range_sum(node, lower, upper):













assert range_sum(Node(7), 1, 100) == 7
assert range_sum(Node(2, Node(3, Node(4))), 2, 4) == 5
assert range_sum(root, 4, 10) == 13 # see example above
```

Task 4: Object-Oriented Programming & Testing 20 Points

Implement two classes Backpack and Item that can be used to plan the next camping trip. A Backpack has a maximum volume (in liters), an Item has a name and a volume (in liters). All these parameters are provided in the constructors. Please note the following example:

```
bp = Backpack(4.0)
bp.pack(Item("water bottle", 0.75))
bp.pack(Item("lighter", 0.05))
bp.current_volume() # 0.755
bp.unpack() # returns Item("lighter", 0.05)
bp.pack(Item("camping tent", 20.0)) # AssertionError!
```

Implement the three Backpack functions from the example: 1) pack stores an Item. Throw an AssertionError in case the available volume is exceeded. 2) current_volume returns the total volume of all packed items. 3) unpack removes and returns the last added Item or None should the backpack be empty.

Note: The implementation does not have to check for incorrect types or invalid values.

Note: You can omit the required import statements for Backpack, Item, and TestCase.

a) Implementation of `Backpack` 10 Points

b) Implementation of `Item` 2 Points

c) Unit Testing 8 Points

Write a test suite for `Backpack` that checks whether an *arbitrary* implementation follows the previous specification. Use the Python `unittest` module and extend `TestCase`. You don't have to be exhaustive, just write one test for the constructor, one for each of the three defined methods, and one that verifies that large items indeed cause an `AssertionError`.

Note: Do not assert more than one property in a test case.

Task 5: Inheritance & Composition 20 Points

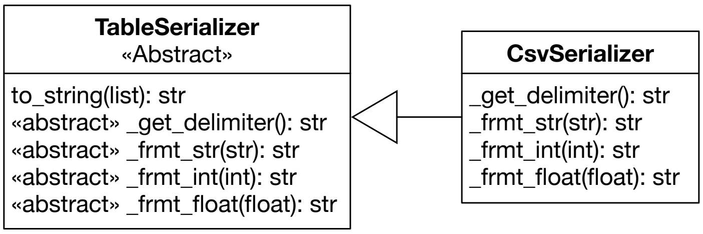
Table-based data has records (rows) with attributes (columns) and is stored in standard formats. One such format is .csv, which contains one record per line and separates attributes by comma. An example is shown in the following Figure. The table is on the left, the .csv representation in the middle, and the programmatic representation (list of tuples) on the right.

Name	Age	Size
Hans	53	1.78
Frieda	27	1.63

```
"Name","Age","Size"
"Hans",53,1.78
"Frieda",27,1.63
```

```
[("Name","Age","Size"),
 ("Hans",53,1.78),
 ("Frieda",27,1.63)]
```

Many variations exist for the formatting: For example, not using a comma, but a tabulator ("\t") as attribute delimiter, avoiding string quotes or using specific number formats. In this task, you will implement a hierarchy around the abstract base class `TableSerializer` that can be extended to support various formats such as the .csv serialization from the example.



TableSerializer
«Abstract»

to_string(list): str
«abstract» _get_delimiter(): str
«abstract» _frmt_str(str): str
«abstract» _frmt_int(int): str
«abstract» _frmt_float(float): str

CsvSerializer

_get_delimiter(): str
_frmt_str(str): str
_frmt_int(int): str
_frmt_float(float): str

The abstract `TableSerializer` has to implement `to_string` that iterates over the table to create the corresponding string representation. To make it extensible, formatting is delegated to the abstract methods `get_delimiter` and `frmt_...`. A subclass has to implement these and can decide that, for example, 1.234 should be represented as `"1.234"` or `1.234` or `1.2` (rounded) or in a different way by implementing `frmt_float` accordingly.

Note: For this task, you can assume that attributes will always be `str`, `int`, or `float`.

Note: `TableSerializer` is an *abstract base class*. Extend `ABC` and annotate abstract methods with `abstractmethod`, to prevent an instantiation of the class. You can omit required `imports`.

a) Implement `TableSerializer` according to the UML specification 10 Points

b) Derive a concrete subclass 8 Points

Implement a subclass, `CsvSerializer`, that follows the .csv format from the example.

c) Using the Hierarchy 2 Points

Instantiate a `CsvSerializer`, format `table`, and `print` it on the terminal.

```
table = (("Name","Age","Size"), ("Hans",53,1.78), ("Frieda",27,1.63))
```

Task 6: Working With Modules 10 Points

You are building a new geo-location-based app that can find the closest train station. Assume there is a library `navigation.py` that provides some useful functions.

```
# content of file "navigation.py"
def get_current_position():
   ’’’Returns the current GPS coordinates (latitude, longitude) in a string like
   "43.63871944444445,-116.2413513485235".’’’
def find_train_stations(position):
   ’’’Given the current position in a tuple of two floats (latitude, longitude),
   returns all stations in a 5km radius. The stations will be returned in a list
   of tuples with the format (str, (float, float)). The first element is the
   station name, the second is the exact position tuple. The values are ordered
   by distance (closest first). The list is empty if no stations are nearby.’’’
```

Implement a function `find_next_station` that finds the closest train station by reusing this library. The function should return the name of the station as a string or return `None` if no stations are nearby. Subsequent calls should always consider the latest position of the user.

Note: Do not forget the `import`s. All files are located at the root of the module search path.

```



def find_next_station():




print(find_next_station()) # for example, "Bahnhof Oerlikon"
```

Useful Python Functions

Strings

str.upper() / str.lower() Returns a new string, in which all letters are converted to *uppercase/lowercase*.

str.isupper() / str.islower() Returns `True` if all characters in the non-empty string `str` are uppercase/lowercase, `False` otherwise.

str.split(sep) Returns a list of the words of the string `str`, separated on occurrences of `sep`. If `sep` is absent or None, the string is separated by whitespace characters (space, tab, newline, return, formfeed).

str.join(words) Returns a string by concatenating the list of words with intervening occurrences of str.

str.isalpha() / str.isdigit() Returns True if all characters of a non-empty string are alphabetic/numeric, `False` otherwise.

str.startswith(prefix) Returns `True` if string `str` starts with `prefix`, `False` otherwise.

str.endswith(suffix) Returns `True` if the string ends with `suffix`, otherwise `False`.

string.find(x) Returns the starting index of `x` if it occurs in the string, otherwise -1.

string.replace(old, new) Returns a copy of the string where all the occurrences of the substring `old` are replaced with the substring `new`.

Lists

list.append(x) Add an item `x` to the end of the list `a`; equivalent to `a[len(a):] = [x]`.

list.remove(x) Remove the first item from the list whose value is `x`. Throws an error if no such item exists.

list.index(x) Returns index of first item in list whose value is `x`. Throws an error if no such item exists.

list.count(x) Counts the occurrences of `x` in a list.

Dictionaries

key in dict Returns `True` if dictionary `dict` has `key`, `False` otherwise.

dict.keys() Returns a list of all keys defined in dictionary `dict`.

dict.items() Returns a list of `dict's` (key, value) tuple pairs.

dict.values() Returns a list of dictionary `dict's` values.

dict.get(key, default=None) Returns the value associated with `key` or `default` if key does not exist.

dict.pop(key) Removes `key` from the dictionary and returns its former value.

Files

open(filename, ‘r’) Opens the file `filename` for reading and returns a file handle.

open(filename, ’w’) Opens the file `filename` for writing and returns a file handle.

f.close() Closes the file handle `f`.

f.readline() Returns the next line of file handle `f`.

f.readlines() Returns all lines of file handle `f`.

os.path.isfile(file) Returns `True` if `file` is an existing regular file.

Other

isinstance(obj, type) Returns `True` if `obj` has a type compatible to `type`, `False` otherwise.

len(obj) Return the length of an object. `obj` may be a sequence (e.g., string, list, etc) or a collection (e.g., dictionary).

sorted(sequence) Return a new sorted list from the items in sequence.

TestCase

assertEqual(a, b) Test that `a` and `b` are equal or fails the test, otherwise.

assertTrue(a) / assertFalse(a) Test that `a` is `True / False`.

assertRaise(Type) Can be used in a `with` statement to make sure that the enclosed code `raises` the given error type. The test fails, if not.