Informatik I
Introduction to Programming

Assessment Exam Winter 2018

General Guidelines:

- It is possible to achieve **90 points**, achievable through completing all tasks.
- You have **90 minutes** to complete the test.
- Please check that you have received all 13 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 can 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 for your answers, you can freely choose between version 2 and 3 and their according functions. 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

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

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. Leave the *value* field empty if the expression does not return any value. In case of errors, state *NoneType* as the type and write *error* as the value.

Note: It is enough to name the simple type without mentioning the module, e.g., *int* or *integer*.

Note: The snippets are invoked in separation and do not have any side effects on each other.

Note: Read the snippets very carefully. Not all answers are obvious.

a) 2 Points

```
5/2 + 3
```

Type: □ Value: □

b) 2 Points

```
b1 = "13579"
b2 = "02468"
b1[5:] + b2[-5]
```

Type: □ Value: □

c) 2 Points

```
(lambda x: x%2 == 0)(2)
```

Type: □ Value: □

d) 2 Points

```
d = [[[1, 1], [2, 2]], [[3, 3], [4, 4]]]
d[0][2]
```

Type: □ Value: □

e) 2 Points

```
class X: pass
class Y(X): pass
class Z(Y): pass
if isinstance(Z(), X):
   e = 1
else:
   e = 2.3
e
```

Type: Value:

f) 2 Points

```
f = sorted({ ’a’:1, ’b’:2, ’c’:3 }.items())
f[0]
```

Type: Value:

g) 2 Points

```
def g():
  return False
"x" if not g else {}
```

Type: Value:

h) 2 Points

```
def addition(arr):
  s = 0
  for el in arr:
    if el % 2 == 0:
      s += el
  return s
addition([1, 2, 3, 4])
```

Type: Value:

i) 2 Points

```
class Animal:
 def talk(self):
   return "Moo!"

class Dog(Animal):
 pass

dog = Dog()
dog.talk()
```

Type: Value:

j) 2 Points

```
class Employee:
   id = 0
   def __init__(self, name):
      self.name = name
      self.id = Employee.id
      Employee.id += 1

emp = Employee("Marc")
emp.id
```

Type: Value:

Task 2: Functions 10 Points

In this task, you will implement several utility functions. The required behavior of your implementations is illustrated with asserts at the end of each subtask.

Note: You do not have to check for None arguments, but you must handle corner cases of the expected argument type (like negative integers or empty strings).

a) Implementation of split 5 Points

Write a function called split that splits a given string on each space character into words and returns them as a list. You can assume that the string only contains letters and single spaces, no punctuation or other special characters.

```
def split(text):




assert split("") == []
assert split("aaa") == ["aaa"]
assert split("a bbb cc") == ["a", "bbb", "cc"]
```

b) Reverse Index 5 Points

Implement the function `rev_idx` that builds and returns a *reverse index* for a list of words: a `dictionary` that contains each word as a `key` and a list as `value`. The list should contain all indexes at which the word appeared in the initial list.

Note: The function should not distinguish upper-case and lower-case letters.

```
def rev_idx(words):

assert rev_idx([]) == {}
assert rev_idx(["a","b"]) == {"a": [0], "b": [1]}
assert rev_idx(["a","B","A","aa"]) == {"a": [0, 2], "aa": [3], "b": [1]}
```

Task 3: Recursion 16 Points

In this task, you will implement several *recursive* utility functions. The required behavior of your implementations is illustrated with `asserts` at the end of each subtask.

a) Product of two numbers 8 Points

Implement a `prod` function that multiplies two numbers *recursively*. You can assume that `x` and `y` are positive integers, but you are not allowed to use the regular multiplication operator `*`.

```
def prod(x, y):






assert prod(2, 0) == 0
assert prod(5, 2) == 10
```

b) Reverse List 8 Points

Implement the `reverse` function that reverses a list *recursively*.

```
def reverse(l):






assert reverse([]) == []
assert reverse([2]) == [2]
assert reverse([2, 6, 5]) == [5, 6, 2]
```

Task 4: Object-Oriented Programming & Testing 20 Points

In this task, you will implement a class and its corresponding unit tests. Use the Python library `unittest` for writing the tests. You do not need to provide any `import` statements.

a) Accounting 8 Points

Define a class `BankAccount`. The account balance can be requested through `balance` and changed with `deposit` and `withdraw`. New accounts have a `balance` of 0, but a credit limit is provided in the constructor. A call to `available` will report the maximum amount that is available for a withdrawal. `raise` an `AssertionError` when a negative credit limit is provided in the constructor or when a `withdraw` exceeds the allowed credit limit.

Note: `deposit` and `withdraw` do not have a return value and do not `print` anything.

```
# example usage
acc = BankAccount(100)
print(acc.balance()) # prints '0'
print(acc.available()) # prints '100'
acc.deposit(30) # balance: 30, available: 130 (illustration, no "print")
acc.withdraw(40) # balance: -10, available: 90 (illustration, no "print")
acc.withdraw(91) # AssertionError
```

b) Black-Box Unit Tests 12 Points

Someone else will provide another `BankAccount` implementation that follows the previous specification. Extend `TestCase` and write a test suite that checks the correctness of this other implementation, without knowing any details about the inner workings. You don't have to be exhaustive, just write one test for the constructor and one for each method of `BankAccount` that check normal usage. In addition, provide a test to check that excessive withdrawals cause an `AssertionError`.

Note: Do not use the built-in `assert` statement of Python. Instead, use the asserts of the `TestCase` base class.

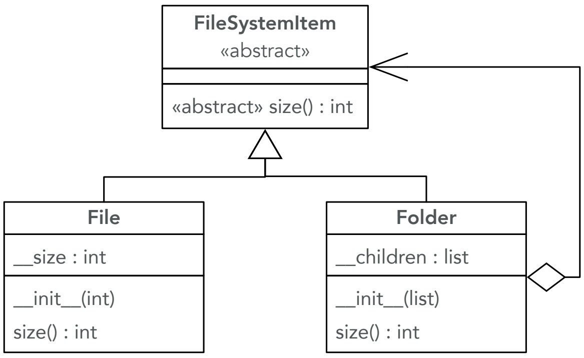
Note: Do not assert more than one property in a test case.

Note: Think of this task as an isolated unit, unrelated to the previous task. We will grade both individually and can be solved without the other.

Task 5: Inheritance & Composition 14 Points

You have decided to implement an abstraction for the file system to make it easier to calculate the required space on a hard disk. In the following, you will find an illustration of this abstraction as a UML diagram together with an example that specifies the usage of the classes that are to be created. The abstraction features a generic `FileSystemItem` and two specializations, `File` and `Folder`.

Every `FileSystemItem` can be asked for its size. The `File` size is static and is provided at instantiation time. The size of a `Folder` can be determined by adding the sizes of its content. This content is provided as a list to the constructor and can contain both `Files` and nested `Folders`.



```
assert File(1).size() == 1
assert Folder([]).size() == 0
assert Folder([File(2), \
  File(3)]).size() == 5
assert Folder([File(4), \
  Folder([File(5)])]).size() == 9
```

In the following, you will implement the three classes `FileSystemItem`, `File`, and `Folder`. You do not need to use `import` and you don't need to check for invalid parameters.

a) Implement the abstract base class `FileSystemItem` 4 Points

Implement the class `FileSystemItem` as an *abstract base class*. Extend `ABC` and annotate the `size` method with `abstractmethod`, to prevent an instantiation of the class.

b) Implement `File` according to the specification. 4 Points

c) Implement `Folder` according to the specification. 6 Points

Task 6: Working With Modules 10 Points

Modern computers contain various sensors that monitor the health state of a system. Assume that you have found the following file `stats.py` that provides a utility functions for accessing many different metrics. You want to integrate one particular function in your program, the header says:

```
# content of file "stats.py"
def get_system_stats():
   '''Provides access to useful system stats (key -> description):
   - cpu_temp -> The temperature of the CPU sensor in "Kelvin" (float)
   - fan_speed -> The speed of the CPU fan in "rotations per minute" (int)
   - ... (several other stats, cut for brevity)
   Returns: A dictionary that contains all stats by key.'''
```

Define a class `TempReader` that makes it easier to access the CPU temperature. The class should have two methods `celsius` and `fahrenheit`, which return a formatted string of the current temperature in the corresponding scale.

Unfortunately, the utility function you found reports temperatures values in Kelvin (T_K), so they have to be converted to Celsius ($T_C = T_K - 273.15$) and Fahrenheit ($T_F = 1.8 \cdot T_K - 459.67$) first. The reported numbers should be rounded to one digit after the comma.

Note: Subsequent calls to `TempReader` methods should always report up-to-date values!

Note: Do not forget to include `import` statements in your solution.

Note: Both files in this task are located at the root of the module search path.

```
# content of file "TempReader.py"




# expected behavior with example output
tr = TempReader()
print(tr.celsius()) # "56.2C"
print(tr.fahrenheit()) # "133.2F"
```

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.

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 there is no such item.

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

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.