University of Zurich[UZH]

s.e.a.l. software evolution & architecture lab

Informatik I
Introduction to Programming

Assessment Exam Winter 2017

General Guidelines:

- It is possible to achieve **90 points**, achievable through completing all tasks.
- You have **90 minutes** to complete the test.
- You must **not** use any additional resources. Non-native speakers may use a dictionary.
- 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.
- 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.

Guidelines 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.
- You **must** sign the German version of this test and give your answers there.
- You can use English as the language for text answers.
- **Answers in this English version of the exam will be ignored.**

Task 1: General Questions — 14 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: You do not need to mention the module for the types, just write the exact name of the type.

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

```
1.2 + 3
```

Type: Value:

b) 2 Points

```
b1 = "13579"
b2 = "02468"
b1[1:3] + b2[-3:-1]
```

Type: Value:

c) 2 Points

```
def c(value):
    return value
c(False)
```

Type: Value:

d) 2 Points

```
d = [1, 2.3, (True, None, "bar")]
d[3][3][3]
```

Type: Value:

e) 2 Points

```
def e():
   for i in { 'a':1, 'b':2, 'c':3 }:
      return i
e()
```

Type: Value:

f) 2 Points

```
f = ((1), (2,3))
f[0]
```

Type: Value:

g) 2 Points

```
class g(object):
   x = 1.2
   def __init__(self):
      self.x = 3
g.x
```

Type: Value:

Task 2: Functions 16 Points

In this task, you will be asked to write some simple utility functions. Each subtask will shortly introduce a problem. We include examples for each subtask that illustrate the expected behavior and that your implementation must satisfy.

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) Convert to String 4 Points

Implement a function that converts integer values to a string. The functions should return "«n» is even" if `n` is an even number, or "«n» is odd" otherwise.

```
def stringify(n):
      # write your implementation here




assert stringify(10) == "10 is even"
assert stringify(5) == "5 is odd"
```

b) Compute Statistics 4 Points

Implement a function that calculates three metrics for a list of integer values: (i) the *second-highest* value, (ii) the *average* value, and (ii) the *median value*. For an odd number of values, the median is the value "in the middle". Otherwise, it is the average of the two central numbers in the list.

```
def compute_stats(lst):
      # write your implementation below




stats = compute_stats([1,12,4,5,8])
assert stats[0] == 8 and stats[1] == 6 and stats[2] == 5
```

c) Count Characters 4 Points

Implement a function that counts the occurrences of upper and lower case letters in a string. The return value is a dictionary that contains the respective values through the keys `upper` and `lower`.

```
def count_letters(s):
      # write your implementation here
```

```
d = count_letters("Abc Defg HiJ!")
assert d["upper"] == 4 and d["lower"] == 6
```

d) Mirror 4 Points

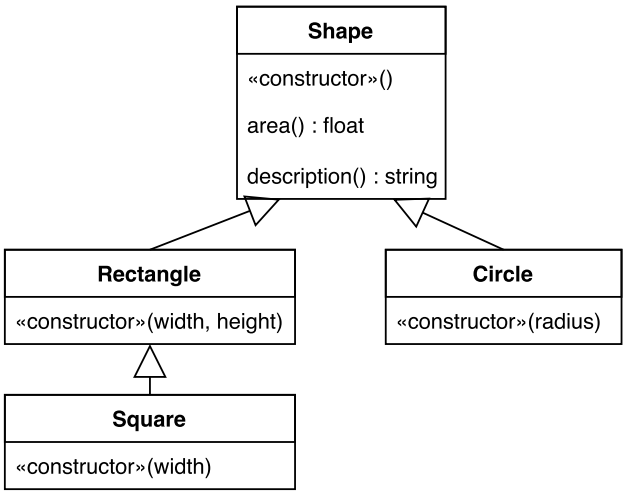
Implement a function that mirrors a string, i.e., it takes the string and adds its reversed form. The last letter should not be duplicated.

```
def mirror(s):
      # write your implementation here
```

```
assert mirror("") == ""
assert mirror("a") == "a"
assert mirror("abc") == "abcba"
```

Task 3: Object Oriented Programming 15 Points

You are working in a project on mathematical software and you have to implement geometric operations. You want to implement a class hierarchy that minimize the amount of code to write by taking advantage of inheritance. Consider the following class diagram for the classes to implement.



The `Shape` class is the base type of the hierarchy. It is possible to calculate the `area` of a `Shape` and to request a textual `description` of any instance. In the following, you will implement the type hierarchy around `Shape` in three subsequent steps.

a) Define Base Class 6 Points

Implement the base class `Shape`. Provide a default implementation for `area` that returns `0`. Introduce an implementation of `description` that works for arbitrary extensions of `Shape`.

Note: Use the snippet `type(o).__name__` to get a simple type name for `o`.

```
# define Shape here
```

```
assert Shape().description() == "Shape with area 0"
assert Shape().area() == 0
```

b) Extend the Base Class 5 Points

Now, implement `Rectangle` and `Circle` classes that extend `Shape`. Make sure that your implementations reuse `description` that is defined in the base class.

Note: The area of a circle with radius r can be calculated with $a = \pi r^2$, approximate π with 3.

```
# define Rectangle and Circle here
```

```
assert Rectangle(2, 5).description() == "Rectangle with area 10"
assert Rectangle(2, 5).area() == 10
assert Circle(5).description() == "Circle with area 75" # Pi=3!
assert Circle(5).area() == 75
```

c) Reuse Implementations 4 Points

Extend the type hierarchy even further with a `Square`, a special `Rectangle` with two equal sides. Make sure that your implementation reuses `area` that is defined in its base class.

```
# define Square here
```

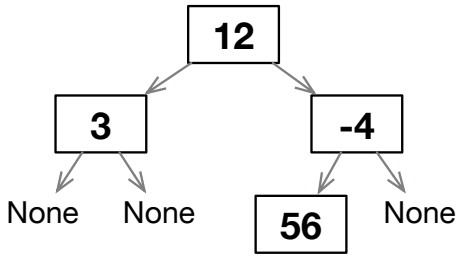
```
assert Square(5).description() == "Square with area 25"
assert Square(5).area() == 25
```

Task 4: Recursion — 20 Points

Python supports the definition of recursive types. Consider the following definition of a `Tree` element that has a value (`val`) and two children `left` and `right`.

```
class Tree(object):
   def __init__(self, val, left = None, right = None):
      assert type(val) == int
      assert left == None or type(left) == Tree
      assert right == None or type(right) == Tree
      self.val = val
      self.left = left
      self.right = right
```

Using `Tree`, it is very easy to instantiate complex tree structures with values. The simple example `Tree(12, Tree(3), Tree(-4, Tree(56)))` would instantiate a structure like this:



In this task, you will write a function that can sum-up all numbers contained in an arbitrary instance of `Tree`. The following test cases illustrate the expected behavior of the function.

```
assert sumTree(None)) == 0
assert sumTree(Tree(1))) == 1
assert sumTree(Tree(1, Tree(2)))) == 3
assert sumTree(Tree(5, Tree(-1), Tree(-2)))) == 2
assert sumTree(Tree(1, Tree(2, Tree(3, Tree(4)))))) == 10
```

On the next page, implement the function with the signature `sumTree(tree)` that returns the sum.

Note: You are not allowed to (and should not need to) introduce new parameters with default values.

```
# implement sumTree here
```

Task 5: Testing — 15 Points

A colleague of you had to implement a function for a cashing machine that is operated with Info1-Dollars, which are only available in 10$, 5$, 2$, and 1$ coins. The function should decide which coins to give out when change is returned. If the total amount of change is less than 3$, no change should be returned at all. It is very important that not only the amount of change is correct, the function should also return it in as few coins as possible.

Your colleague designed the function with the signature `change(int) : tuple`, in which the change amount is provided as an argument and the coin selection is returned as a tuple. For illustration, consider changing 23$ (`change(23)`), which results in the tuple `(2,0,1,1)` ($2 \times 10\$ + 0 \times 5\$ + 1 \times 2\$ + 1 \times 1\$ = 23\$$). The function is always only called with `int` values, but it needs to handle invalid values. The function should return `None`, if no change needs to be returned or if a negative (invalid) change amount is requested.

In this task, you don't need to redefine `change`. Instead, your task is to write *black box* tests that assure the correctness of the function, without knowing its actual implementation.

Note: You can use the function `amount(tuple):int` in your tests to calculate the amount represented by a tuple (e.g., `amount((2,0,1,1))` is `23`). This function does not need to be tested.

Note: You should follow the style of task 4 (`treeSum`) and use `assert` in your tests to check for individual values.

a) Validate Change Amount — 4 Points

Validate for all values of up to 100$ that the change *amount* is correct.

b) Validate Coin Selection — 6 Points

Test meaningful corner cases to make sure that the *selection* of coins is correct.

Task 6: Modules and Git 10 Points

a) Accessing Python Modules 6 Points

One of your systems crashed and you have to investigate what happened. You can use a Python module that is implemented in the file `logs.py` that helps you reading the log files of the system. The module provides the function `get_logs`, which is documented as follows:

```
def get_logs():
    '''
    A utility function for accessing today's system log.
    Returns:
        A list that contains the strings of all logged messages.
    '''
```

The log contains all messages that have been printed during the current day. Every line in the console logs corresponds to a message and starts with a prefix, for example:

```
INFO: Startup Error
ERROR: StaleData
DEBUG: User Registered
DEBUG: Interaction failed...
```

Implement the function `filter_logs` that can find all logged messages of today that contain a specific text. Use the module `logs` to fetch the system log and return only the matching lines. The case of the letters should not matter for the matching, but should be preserved for the output.

```
# import

def filter_logs(text):

print(filter_logs("ErRoR")) # ['INFO: Startup Error', 'ERROR: StaleData']
```

b) Git Usage 4 Points

You are working in a project that uses Git to version control its files. Assume that you are currently looking at a terminal window. Your current working directory is the folder that contains the repository contents and the repository is fully set-up: your only `remote` is set to `origin` and you have checked-out the `master` branch right now.

Follow a typical Git workflow now to create a new feature:

- Create a new branch `new_feature`.
- Work on the file `f.py`.
- *Add* and *commit* the file, use "added f.py" as the commit message.
- *Push* your new branch to the remote.

List all the necessary Git instructions to perform these actions. State "`working on file`" to indicate when you create/change the file.

Useful Python Functions

Strings

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

dict.has_key(key) 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.