NUI Galway
OÉ Gaillimh

Semester I Examinations 2020/2021
Solutions/Grading Scheme

Exam Code(s)	1MAI1, 1CSD1, 1SPE1, 2SPE1
Exam(s)	MSc in Computer Science (Artificial Intelligence), MSc in Computer Science (Artificial Intelligence) - Online
Module Code(s)	CT5132, CT5148
Module(s)	**Programming and Tools for Artificial Intelligence**
Discipline	School of Computer Science
Paper No.	1
External Examiner	Prof. Pier Luca Lanzi
Internal Examiner(s)	Prof. Michael Madden
	Dr. James McDermott *
Programme Coordinator(s)	Dr. Matthias Nickles, Dr. James McDermott
Instructions	**Answer all 4 questions. All are worth equal marks.** You may answer either: in a Word document or similar, and then `CT5132_CT5148_PTAI_Answer_Sheet.docx` is suggested; or on paper, uploading a scan of the pages. This is an **open-book** exam: you may **read** textbooks, notes, and **existing resources** on the internet. You may **not communicate** with anyone, in person, via phone, internet, or otherwise. You may **not post questions** on internet sites or elsewhere during the exam.
Duration	2 Hours exam plus 30 minutes for upload
Number of pages	11 (including this page)
Discipline	Computer Science
Requirements	
Release in Exam Venue	Yes ☒ No ☐
Release to Library	Yes ☒ No ☐

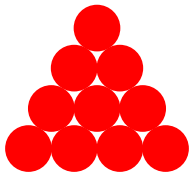
Question 1: Basic Python

(a) Distinguish between a function that prints its result and a function that returns its result. What are the reasons for choosing one over the other? **[5]**

Solution. A function that returns a result doesn't print anything. [A function that prints its result and doesn't return anything has a None return value.] The former is more flexible, as the caller can print the result if needed, but not the other way around.

Grading. 2+3.

(b) Your friend wants to calculate the number of snooker balls in a triangular arrangement, where the last row is of length n, as shown. They have found some good code on the internet, but accidentally discarded indentation while copy-pasting. Fix this. **[5]**



```
def snooker(n):
    if n == 0:
        return 0
    else:
        return n + snooker(n - 1)
n = 5
m = snooker(n)
print(m)
```

Grading. 2 for fixing the function, 3 for fixing the call. Common error: putting the call inside the function.

(c) Your friend has written some code which calculates the number of followers for several Twitter accounts. It stores the result in a list, alternating account name and follower count, e.g.:

```
['StephenFry', 1000, 'BillGates', 2000, 'NYT', 1500]
```

Criticise this choice of data structure and suggest an alternative. **[5]**

Solution. This is inconvenient, because when using this data structure we'll need to manually deal with the alternation, e.g. using a mod operator. The list is of mixed type. If we don't maintain the alternation, all our logic will break. A dict would be better, where the key is the account name and the value is the follower count.

Grading. 2 for criticism involving indices/types. 2 for "dict", and last 1 for full spec key:value.

(d) With reference to the variable `x` below, explain *duck typing* in Python. **[5]**

```
def count_zero(x):
    c = 0
    for a in x:
        if a == 0:
            c += 1
    return c
```

Solution. Duck typing: Python assumes that the types will work, rather than checking. Here, x is assumed to be iterable (for a in x) and every element (a) is assumed to be comparable to an integer. If so, everything else works. If not, an exception is raised. This is flexible – we can pass in various types such as x a list of integers, or x a set of integers, without worrying or rewriting our function.

Grading. 2: Idea that x may have different types – flexibility. 1 for what types will work. 2 for complete idea – it is never checked, just try it.

(e) What is the difference between these three pieces of code? Which is correct and why? **[5]**

```
import random
results = []
for repetition in range(100):
    random.seed(0)
    results.append(run_experiment())
```

```
import random
results = []
random.seed(0)
for repetition in range(100):
    results.append(run_experiment())
```

```
import random
results = []
for repetition in range(100):
    random.seed(repetition)
    results.append(run_experiment())
```

Solution. We can assume this is a typical experimental situation where we have a randomised algorithm (eg training an ML model) and we want to run multiple times for a large statistical sample, but we want the experiment to be replicable.

The first snippet is incorrect, as it runs the same experiment (same seed) 100 times. The second runs 100 distinct experiments, but the individual runs can't be easily replicated. The third is correct as we get 100 independent runs, and we can re-run any of them.

Common errors. Failing to state differences between them.

Grading. 2 for running multiple times, each controllable. 2+1 for explanations/distinction of the bad ones.

Question 2: Advanced Python

(a) What does this regular expression do? Explain with the help of some example strings. **[5]**

```
__\w*__\((.*)\):
```

Solution. It matches Python double-underscore function definitions, and extracts the arguments using the grouping. Eg it will match `def __init__(self, n):`, and will extract `self, n`. It will not match `def factorial(n)` as that lacks the double-underscores.

Grading. 3 for main part (explanation plus example 2+1); 2 for grouping.

Common errors. Missing the grouping parentheses.

(b) In the following code, at the line marked `Line 4`, which names are in scope: `random`, `randrange`, `main`, `f`, `x`, `y`, `z`? **[5]**

```
from random import randrange
def f(y):
    y = y ** 2
    # Line 4
    return y
z = f(2)
def main():
    x = 3
    print(f(x))
```

Grading. y, f, randrange are in scope, the others not. Lose 1 for each incorrect.

(c) Would the function `f` in part (b) be suitable for *memoisation*? Why or why not? **[5]**

Solution. No: it would not break (it's not actually randomised) but it's not long-running so no saving.

Grading. 3 for why not, 2 for not breaking anything.

Common errors. A lot of people think memoisation applies only to recursive functions, which is not correct.

(d) Suppose we are writing a networking program which stores its configuration in a `dict`, e.g.:

```
{'n_connections': 100, 'port': 8080}
```

Show how we can store such a configuration on disk and later read it back in to a `dict`, using `eval`. **[5]**

```
d = {'n_connections': 100, 'port': 8080}
open("config.dat", "w").write(str(d))
# later
d = eval(open("config.dat").read())
```

Grading. 2 for saving, 3 for reading.

Common errors. Using eg `pickle` is a viable solution here and easily Googled, but the question specifies `eval`.

(e) What is the time complexity of the following function, with respect to the length of `text`? What can we say about complexity with respect to the length of `targets`? **[5]**

```
def count_letters(text, targets):
    # text is a long string, eg a book
    # targets is a list of letters a-z and A-Z which we wish to count
    count = 0
    for c in text: # one character at a time
        for t in targets:
            count += 1
    return count
```

Grading. 2 for O(n) in text. 1 for O(n) in targets. 2 for seeing targets is limited in size so irrelevant.

Question 3: Data Science

(a) Given the array `a` below, write an array `b` such that `a * b` gives the result shown. Explain in your own words the rules for Numpy broadcasting using this example. **[5]**

```
a = np.array([[[ 1,  2,  3],
               [ 4,  5,  6]],

              [[ 7,  8,  9],
               [10, 11, 12]]])

# b = ?
a * b

array([[[  10,   20,   30],
        [ 400,  500,  600]],

       [[  70,   80,   90],
        [1000, 1100, 1200]]])
```

Solution. b = np.array([[10], [100]]). We take the shapes of a and b. Each shape is a tuple and they may be different lengths: here (2, 2, 3) and (2, 1). We right-align them. Then working from the right, each pair of values is compatible if (1) identical, or (2) one is equal to 1, or (3) one is missing. The right-most values are 3 and 1 (ok); then 2 and 2 (ok); then 2 and missing (ok).

Grading. 2 for correct array b. Other expressions for b are also possible, with redundancy. 1 for rules, but 2 for application of rules in this example.

Common errors. Providing b but no explanation. Missing the right-align step.

(b) In your own words, explain the significance of *whether or not* a Scikit-Learn `estimator` object contains a `_coef` variable (or another variable named with a leading underscore). **[5]**

Solution. Error in question (should be trailing underscore, not leading). If it has been fitted/trained, then it has the variable `coef_` or similar, else it doesn't.

Grading. Due to error, allow full marks to all.

(c) Rewrite the following R code using the `magrittr` pipe, where `D` is a tibble and `a` and `b` are columns. **[5]**

```
select(mutate(D, a=0.4*a, b=0.4*b), year, a, b)
```

Solution.

```
D %>% mutate(a=0.4*a, b=0.4*b) %>% select(year, a, b)
```

Grading. 2 for any kind of unwind/rewrite, 1 for still using mutate and select, 2 for fully correct.

(d) The following data is not *tidy*. Explain why not, and show what it would look like in tidy format. **[5]**

Parameters	n=3,m=3	n=3,m=4	n=4,m=3	n=4,m=4
Elapsed Time	122	150	143	190
Accuracy	90	95	87	91

This is not tidy because each row is storing a variable, not an observation; because each column is storing an observation, not a row; and because each column name contains multiple parameter values.

n	m	time	accuracy
3	3	122	90
3	4	150	95
4	3	143	87
4	4	190	91

Grading. 1 for ideas of tidy data. 2 for specific application of tidy idea in this context, 2 for rewrite. Not required to write code here.

Common errors. Keeping n and m together, instead of separate columns.

(e) In the following data we wish to find the sum of sales per location. Using this example, explain the *group-by* mechanism (which exists in both R and Pandas). You don't need to write code or carry out calculations. **[5]**

Seller	Location	Day	Sales
A	Galway	Mon	1000
A	Galway	Tue	1100
A	Limerick	Mon	1200
A	Limerick	Tue	800
B	Galway	Mon	1100
B	Galway	Tue	1400
B	Limerick	Mon	1400
B	Limerick	Tue	1300
C	Galway	Mon	900
C	Galway	Tue	1000
C	Limerick	Mon	1000
C	Limerick	Tue	1500

Group-by requires an argument specifying which column to group by. It then splits the table into new sub-tables, where each sub-table corresponds to one possible value of that column,

and contains all rows with that value. For example, we'll have a new sub-table with all the Galway rows, and another with all the Limerick rows. Any aggregation function is then applied per sub-table, eg calculating the sum of Sales across Galway rows, and separately across the Limerick rows. This gives one new summed row for each original value in the column. Finally, these summed rows are combined.

Grading. 2 for groupby understanding, 3 for explanation in this specific context.

Question 4: Tools and Applications

(a) State the 4-grams in the following bitstring: `001000100011`. **[5]**

0010, 0100, 1000, 0001, 0010, 0100, 1000, 0001, 0011.

Grading. 2 for getting one right, 2 for including the overlaps, 1 for all. Ok to remove duplicates.

Common errors. Missing the overlaps, ie providing 0010, 0010, 0011 only.

(b) Consider the grammar below, where `<expr>` is the start symbol. Show that it can generate some valid Python expressions, and can also generate some invalid ones. **[10]**

```
<expr>  ::= {<kvs>} | {<vs>}
<kvs>   ::= <key>: <val> | <key>: <val>, <kvs>
<vs>    ::= <val> | <val>, <vs> | <kvs>
<key>   ::= 'a' | 'b' | 'c' | 'd'
<val>   ::= 1 | 2 | 3 | 4 | 5 | 6 | 7
```

Solution

We can generate a valid dict, eg:

```
<expr> -> {<kvs>}
       -> {<key>: <val>}
       -> {'a': <val>}
       -> {'a': 1}
```

We can generate a valid set, eg:

```
<expr> -> {<vs>}
       -> {<val>, <vs>}
       -> {2, <vs>}
       -> {2, <val>}
       -> {2, 3}
```

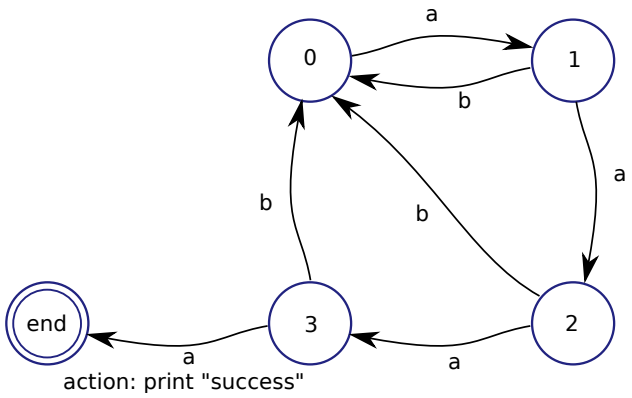
We can generate an invalid dict/set hybrid expression, eg:

```
<expr> -> {<vs>}
       -> {<val>, <vs>}
       -> {7, <vs>}
       -> {7, <kvs>}
       -> {7, <key>: <val>}
       -> {7, 'b': <val>}
       -> {7, 'b': 4}
```

Grading. 3 + 2 for a valid one with derivation, 3 + 2 for an invalid one with derivation.

Common errors. Skipping steps in derivation.

(c) In the finite state machine (FSM) shown below, the start state is state 0 and the end state is as indicated. Edges are labelled with inputs, and one edge also has an action as shown. Explain what this FSM does, with the help of some example inputs. Encode this FSM as Python code, either in the *State, input, state, action (SISA)* format, or another unambiguous format if you prefer. **[10]**



Solution. Possible inputs to this FSM are 'a' and 'b'. Any 'b' will lead to state 0. So, eg, aabbababaaaa will move through various states, but will reach the end state at the end of this input. The sequence abab will move from 0 to 1 to 0 to 1 to 0. A sequence of 4 consecutive 'a' inputs will lead to the end state, printing "success". We could say that this FSM computes *whether there is a string of four consecutive 'a's in the input.*

```
SISAs = {
  (0, 'a'): (1, None),
  (1, 'a'): (2, None),
  (1, 'b'): (0, None),
  (2, 'a'): (3, None),
  (2, 'b'): (0, None),
  (3, 'a'): ('end', lambda: print("success")),
  (3, 'b'): (0, None)
  }
```

Grading. 3 for explanation, 3 for example inputs. 4 for perfect encoding, but just 2 for a partial encoding.

Common errors. Omitting SISAs, omitting example inputs.