CGS600A: Computational Tools in Cognitive Science

Midsem test

Max marks:100
Time:2 hours 18 Sept. 2018

1. *Answer all 3 questions. The paper has 4 pages.*
2. *Please start the answer to a question on a fresh page and answer all parts of a question together.*
3. *You can only refer to your own handwritten class notes and nothing else.*
4. *Where needed assume that code fragments are typed into the Python interpreter.*
5. *Any code and contents of a file are shown using typewriter font like* `this`.
6. *If you are making any assumptions clearly state them.*
7. **Mere correctness is not enough. Quality of solution, code will carry credit.**
8. *If you think executing the code will lead to an error or an exception then indicate the error/ exception.*

1. This question has 8 short answer parts.

   (a) In informal writing dates are often written as follows. For example 9th Sept. 2018 may be written as:

   `9/9/18 or 9-9-18 or 9.9.18`

   Note that the date is not in the dd-mm-yyyy format. Essentially, the minimum number of characters required to express the date are used. Assuming all dates are in the same century (assume 01 to 99 - so yy is enough) write a regular expression that will match all correct dates in informal writing.
   Disregard leap years and assume Feb. has only 28 days.
   Use definitions for sub-expressions, where needed, for clarity.

   Solution:

```
d28 = [1-9]|1[1-9]|2[1-9]
d30 = d28|29|30
d31 = d30|31
m30 = 04|06|09|11
m31 = 01|03|05|07|08|10|12
yr = 0[1-9]|[1-9][1-9]
sep = [./-]
date = (d28 (sep) 02|d30 (sep) m30|d31 (sep) m31) \2 yr
```

   Note the use of `\2` to ensure that the two separators between day, month and month, year are the same.

   (b) Write a one line statement that exchanges the values of variable x and y.

Solution:

```
x,y=(y,x)
```

(c) What will be printed when the following code fragment is executed:

```
x=[1,2,3]
y='abc'
def fn1():
    x1=x.append(4)
    y1=y.upper()#upper() changes case of each character
    print(x1)
    print(y1)

fn1()
print(x)
print(y)
```

Solution:

Note that appending by `append` to a list changes the list but does not return a value i.e. returns None.

```
None
ABC
[1, 2, 3, 4]
abc
```

(d) Now suppose the variable name `x1` is replaced everwhere by `x` in the code fragment in part (c) then what is printed out when the changed fragment is executed?

Solution:

In function `fn1` `x` appears on the left hand side so it will be treated as a local variable in `fn1`. Also, it is being used on the right hand side in an expression. But the local variable `x` does not have a value when the expression is evaluated since it has not yet been assigned so we will get a run-time error. The error will occur as the first statement - `x=x.append(4)` is executed. The exact error is:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

(e) What will be the output of the following Python code fragment (clearly indicate blank lines, if any) where the file input.txt is shown after the code fragment:

```
with open('input.txt') as inp:
    for x in inp:
        print(x)
```

The file input.txt:

```
First line
Second line
Third
```

Solution:

Note that a file iterator iterates over lines in the file and reads a line and returns it as a string (actually uses `readline()`) which includes the `\n` as part of the string. `print` prints a newline character at the end after it has finished printing its arguments. So, an extra blank line will be printed after each line in the file `input.txt`.

```
First line

Second line

Third
```

(f) Modify the code in part (e) so that the contents of the file input.txt is exactly duplicated in a file called output.txt. Don't write totally new code make the minimal changes in the code in part (e).

Solution:

`write()` just prints the single string argument. So, the code below will exactly reproduce the file `input.txt`.

```
with open('input.txt') as inp, open('output.txt','w') as out:
    for x in inp:
        out.write(x)
```

(g) What will be the output of the following code fragment:

```
x=(2,3,-4,1)
y=(3,2,1,-4)
sum(u*v for u, v in zip(x,y))
```

What is the code doing?

Solution:

```
4
```

The code is computing the dot/scalar product of the two tuples `x, y`. That is calculating $\sum_{i=0}^{n-1} x[i]y[i]$ where n is the length of a tuple.

(h) Study the code below and answer the questions that follow:

```
class Cls:
    def __init__(self, x, y):
        super().__init__()
        self.x=x
        self.y=y

    def g(self):
        yield self.x
        s=self.x
        while True:
            s=s+self.y
```

```
            if s>15:
                raise GeneratorExit
            yield s

def testCls():
    c=Cls(8, -5).g()
    try:
        for i in range(2):
            print(next(c))
    except GeneratorExit:
        print('Done')
    else:
        print('Success')
```

What is the class `Cls` implementing? Describe as concisely as possible. What will be the output when `testCls()` is executed?

Solution:

The class `Cls` is implementing a potentially infinite arithmetic progression, bounded from above by 15, where each element of the progression can be generated one at a time using the generator function `g`.

```
8
3
Success
```

[8×5 = 40]

2. Answer the questions following the code below after the code fragment has been executed.

```
class ClsA:
   def __init__(self,x,y):
       self.x=x
       self.y=y

   def f(self,z):
       self.x=self.x+z.x
       self.y=self.y+z.y

class ClsB(ClsA):
    def f(self,z):
        self.x=self.x*z.x
        self.y=self.y*z.y

c1=ClsA(3,4)
c2=ClsB(2,3)
c3=ClsA(7,8)
c1.f(c3)
c2.f(c3)
```

(a) What are the values of `c1.x, c1.y, c2.x, c2.y, c3.y`?

Solution:
Note that ClsB does not have an `__init__()` defined in the class itself so it will inherit the `__init__()` from ClsA so c2 will have data attributes x and y.
`c1.x=10, c1.y=12, c2.x=14, c2.y=24, c3.y=8)`

(b) Write the statement that will invoke the function `f` in class `ClsA` on object `c2` with argument `c3`. What will be the values of `c2.x, c2.y` after the statement has executed?

Solution:
`ClsA.f(c2,c3)` or `super(ClsB,c2).f(c3)`

`c2.x=21, c2.y=32`

(c) Assuming namespaces are implemented as dictionaries. Write down the name spaces of `c1, c2` after (b) has executed.

Solution:
Namespace of c1 is: `{'x':10, 'y':12}`

Namespace of c2 is: `{'x':21, 'y':32}`

(d) Write the statement that will allow access to the `x` attribute of `c2` that is inherited from `ClsA`. What will be the output when the statement is executed (after part (b))?

Solution:
`ClsB` is not overriding any attributes of `ClsA` so the attribute x is directly available. `c2.x` will access the value of the attribute. The value is 21.

[5, (3,2), (3,3), (3,1) = 20]

3. Consider the binary tree in Figure 1.

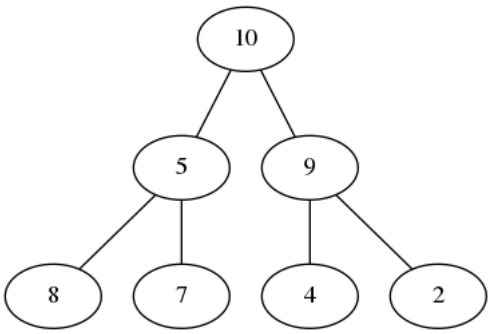

Figure 1: Binary tree for question 3.

We can see that it is recursively made up of a left binary sub-tree, a node value and a right binary sub-tree. Whenever a left and/or right binary sub-tree does not exist - for example for the leaf nodes with values 8, 7, 4, 2 - we set the value to `None` - not shown in the figure.

(a) We define class `BinTree` below with missing code to implement a binary tree. You have to fill in the code to complete the definition of the class. Missing code is indicated by: ... . You are given that the binary tree at node value 8 can be constructed by the expression:
`BinTree(None, 8, None)`.
Fill in the missing parts of code. Basically, you have to define the functions: `__init__` and the getter and setter functions.

```
class BinTree:
    def __init__(self, ...):
        ...
    def getltree(...):# left sub-tree
        ...
    def getrtree(...):# right sub-tree
       ...
    def getnv(...):# node value
        ...
    def setltree(...):
        ...
    def setrtree(...):
        ...
    def setnv(...):
        ...
```

Solution:

```
class BinTree:
    def __init__(self, l, n, r):
    # the data variables should be private
    # get, set methods provided
        self._lbt=l# left binary sub-tree
        self._nv=n# node value
        self._rbt=r# right binary sub-tree

    def getltree(self):
        return self._lbt

    def getrtree(self):
        return self._rbt

    def getnv(self):
        return self._nv

    def setltree(self, tr):
        self._lbt=tr

    def setrtree(self, tr):
        self._rbt=tr

    def setnv(self, n):
        self._nv=n
```

(b) Write an expression that will construct the empty binary tree.

Solution:
`BinTree(None, None, None)`

(c) Write the expression that will construct the binary tree in Figure 1.

Solution:
BinTree(BinTree(BinTree(None, 8, None), 5, BinTree(None, 7, None)), 10, BinTree(BinTree(None, 4, None), 9, BinTree(None, 2, None)))

(d) Suppose we want to iterate over a binary tree then there are many differt ways in which the elements in the tree can be returned. For example, consider traversing the tree as follows:

> At any node first go to the left child. If there is no left child or the node value has not already been returned then return the node value. If the node value has already been returned then go the right child and repeat the process.
> If both left and right children are done then go back to the parent and continue the process till all node values have been returned.

We can write this as `l-n-r` for left, node, right. One can have other ways to traverse the tree for example `n-l-r` or `l-r-n` and still others.

For the tree in Figure 1 if we traverse the tree using `l-n-r` we get the following order for the returned node values:
`8 5 7 10 4 9 2`
Write code for the `__iter__()` and `__next__()` functions to be added in class BinTree that will allow us to iterate over a binary tree and return the nodes in the `l-n-r` order.
Hint: You may requre the `yield from <expression>` *statement.*

Solution:

```
def __iter__(self):
    def nextfn(self):
        if self!=None:
            yield from nextfn(self.getltree())
            yield self.getnv()
            yield from nextfn(self.getrtree())
    return nextfn(self)

def __next__(self):
    return next(self)
```

(e) How will you implement iterators for multiple types of traversals in `BinTree`? Don't write code just give the approach.

Solution:
There are a number of ways to do this:

1. Write separate generator functions as attributes in BinTree for each traversal type. This will mean that `__iter__()` and `__next__()` protocol cannot be used and a user has to explicitly create a generator during looping. But this is a clean solution and the overhead of creating a generator is very little. For example: `for i in gen_nlr(btree)`.
2. A second way is to have a data attribute in BinTree which is set to the traversal type required and let `__iter__()` generate the appropriate generator. This will allow the normal iteration protocol via `__iter__()` and `__next__()` work. But the data attribute has to be set to the right traversal type before each iteration.
3. A third way is a generalization of the first one above. Have a single generator that takes two arguments - the container and the traversal type - and generates the appropriate generator based on the second argument at run time.

The first and third are better than the second option since the traversal type is external to the container and should not be an attribute. If there are very few traversal types then the first is best if there are many traversal types then the third is best though the code for the third option is more complex since the generator function has to be constructed dynamically at run time.

[(6,6), 3, 6, 14, 5 = 40]