CSCI2202: Mid-Term Review

Finlay Maguire (finlay.maguire@dal.ca)
TA: Ehsan Baratnezhad (ethan.b@dal.ca)
TA: Precious Osadebamwen (precious.osadebamwen@dal.ca)

Object Types

Q1. What is the type of each of the following expressions?

```
'5'
5.0
5
[]
[5]
[5,10,15]
(5,10)
{5: 10}
{}
lambda x: x +5
```

Object Types

Q1. What is the type of each of the following expressions?

```
'5'                - string
5.0                - float
5                  - int
[]                 - list
[5]                - list
[5,10,15]          - list
(5,10)             - tuple
{5: 10}            - dict
{}                 - dict
lambda x: x +5     - function
```

Contents of an object don't change its type (although can impact whether it can be casted to another type e.g., `str('5')` vs `str('a')`)

{} can be confusing as python has also has `sets` (defined as `{1,2,3}`) which acts like an unordered mutable tuple. Empty `{}` is always a dictionary though.

Mutability/Immutability

Q2. Which of the following types are mutable and which are immutable?

```
Integers (int)
Strings (str)
Floating point numbers (float)
Lists (list)
Tuples (tuple)
Dictionaries (dict)
Boolean (bool)
```

Mutability/Immutability

Q2. Which of the following types are mutable and which are immutable?

```
Integers (int) - immutable
Strings (str) - immutable
Floating point numbers (float) - immutable
Lists (list) - mutable
Tuples (tuple) - immutable
Dictionaries (dict) - mutable
Boolean (bool) - immutable
```

Mutability is changing part of a value NOT overwriting variable with a new value

“Single” value types are generally immutable (int, float, bool, str)

“Compound” types are generally mutable (list, dict).

Tuples are exception as immutable compound types

Boolean Statements

Q3. What is the boolean value of the following?

```
"hello" in "hello world”
len([1,2,3]) == 4
3 > 2 > 1
[] == False
bool("False")
0.0 or True
False and False or True
```

Boolean Statements

Q3. What is the boolean value of the following?

```
"hello" in "hello world" -> True
len([1,2,3]) == 4 -> 3 == 4 -> False
3 > 2 > 1 -> True
[] == False -> False
bool("False") -> True
0.0 or True -> True
False and False or True -> False or True -> True
```

Breakdown complex booleans into their parts

Even empty lists, tuples, dicts don't resolve to False

And: both sides true to return true
Or: one side true to return true

Operator Precedence

Q4. What does z contain?

```
z = 2 + 2 * 2 ** 2
```

Operator Precedence

Q4. What does z contain?

```
z = 2 + 2 * 2 ** 2

z = 2 + 2 * 4
z = 2 + 8
z = 10
```

Just like normal maths there is an order of precedence in operators.

Resolution order:

{()} > {**} > {*, /, //} > {+, -}

Operations within parentheses resolved first.

Aliasing and mutability

Q5. What is the value of x, y, and z after this code finishes?

```
x = ['a', 'b']
y = x
z = x + ['c']
y.append('c')
```

Aliasing and mutability

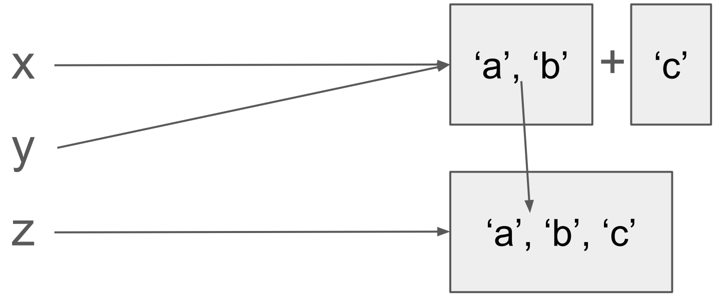
Q5. What is the value of x, y, and z after this code finishes?

```
x = ['a', 'b']
y = x
z = x + ['c']
y.append('c')

x = [‘a’, ‘b’]
y -> x # y and x will be the same
y.append(‘c’)
x = [‘a’, ‘b’, ‘c’]
y = [‘a’, ‘b’, ‘c’]
z = [‘a’, ‘b’] + [‘c’] = [‘a’, ‘b’, ‘c’]
```

Aliasing means multiple names refer to the SAME object

append changes in place



Operator with different types

Q6. What will this code print?

```
print(3 * [1, 2, 3])
```

Operator with different types

Q6. What will this code print?

```
print(3 * [1, 2, 3])
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

This is about remembering how operators work with different types.

* with an integer and list will copy and concatenate the list the integer number of times

Nested loops

Q7. What does the following code print?

```
for x in range(3):
    for y in range(2):
        print(x, y)
```

Nested loops

Q7. What does the following code print?

```
for x in range(3):
    for y in range(2):
        print(x, y)
#x y
 0 0
 0 1
 1 0
 1 1
 2 0
 2 1
```

`range` by default goes from 0 to but not including argument

Outerloop will print: 0, 1, 2
Interloop will print: 0, 1

Nesting means inner loop will run entirely for each iteration of the outer loop.

Break/Continue

Q8. What will each of the following output? Why?

```
for i in range(3):
    for j in range(3):
        if j == 1:
            break
        print(i, j)



for i in range(3):
    for j in range(3):
        if j == 1:
            continue
        print(i, j)

1
```

Break/Continue

Q8. What will each of the following output? Why?

```
for i in range(3):
   for j in range(3):
      if j == 1:
          break
      print(i, j)
0 0
1 0
2 0 # stop running when j is 1
for i in range(3):
   for j in range(3):
      if j == 1:
          continue
      print(i, j)
0 0 # skip printing when j is 1
0 2
1 0
1 2
2 0
2 2 d
1
```

When code reaches:

- `break` exits current loop
- `continue` jump to next iteration

Nesting: inner loop will run for each iteration of the outer loop.

Block A will stop the inner loop running when j is 1 so not get to j > 1

Block B will just skip printing in inner loop when j is 1 so keep running when j > 1

Recursion

Q9. What does rec print(3) print?

```python
def rec_print(x):
    print(x, end=' ')
    if x > 1:
        rec_print(x - 1)
        rec_print(x - 1)
```

Recursion

Q9. What does rec print(3) print?

```
def rec_print(x):
    print(x, end=' ')
    if x > 1:
        rec_print(x - 1)
        rec_print(x - 1)

# r(3) -> r(2) -> r(1)
               -> r(1)
       -> r(2) -> r(1)
               -> r(1)

# r3 r2_1 r1 r1 r2_2 r1 r1
   3   2   1  1   2   1  1
```

Gotchas:

- *var only updated if reassigned*
- *> and >= are different*
- *print kwarg end=' ' means space not newline*

Break down cases i.e.,

- **recursive** case:
  call function again twice with x - 1
- **base case** (x <= 1):
  print(x) and stop/return

recursive case for x = 3 and x = 2 (twice)
base case for x = 1

Keyword and positional arguments

Q10 For this code, what do each of these inputs result in?

```python
def f(x, y=3, z=5):
    print(x + y + z)
f()
f(1)
f(1, 2)
f(1, x=2)
f(1, y=2)
f(1, z=2)
f(1, 2, 3)
f(1, 2, z=3)
f(1, 2, y=3)
```

Keyword and positional arguments

Q10 For this code, what do each of these inputs result in?

```
def f(x, y=3, z=5):
    print(x + y + z)
f()  # TypeError no x argument
f(1) # 1 + 3 + 5 = 9
f(1, 2) # 1 + 2 + 5 = 8
f(1, x=2) # TypeError as x is supplied twice
f(1, y=2) # 1 + 2 + 5 = 8
f(1, z=2) # 1 + 3 + 2 = 6
f(1, 2, 3) # 1 + 2 + 3 = 6
f(1, 2, z=3) # 1 + 2 + 3 = 6
f(1, 2, y=3) # TypeError as y supplied twice
```

MUST have an argument for **x**

Cannot supply same variable more than once

If y or z is not supplied default kwarg value used

Positional args will replace args in order listed

* and ** unpacking can be done

Lazily zipping tuples

Q11. What does the following code return?

```
list(zip((1, 2), (3, 4)))
```

Lazily zipping tuples

Q11. What does the following code return?

```
list(zip((1, 2), (3, 4)))

# (1, 2)
#  |  |
# (3, 4)

# lazy eval -> list

[(1, 3), (2, 4)]
```

`zip` will lazily create tuples out of all the items in the same position across its input iterables i.e., 0th item in each tuple, 1st items… etc

list forces zip create each new tuple and puts the new tuples into a list

Comprehensions

Q12. What will this code print?

```
print(sum([2 * x if x % 2 == 0 else 3 * x for x
in [2, 3]]))
```

Comprehensions

Q12. What will this code print?

```
print(sum([2 * x if x % 2 == 0 else 3 * x for x
in [2, 3]]))

# x = 2  ->  x % 2 == 0 is True  -> 2 * x -> 4
# x = 3  ->  x & 2 == 0 is False -> 3 * x -> 9
# print(sum([4, 9]) -> sum of 4 and 9 = 13
# print(13)
new_list = []
for x in [2, 3]:
     if x % 2 == 0:
          new_list.append(2 * x)
     else:
          new_list.append(3 * x)
print(sum(new_list))
13
```

You can rewrite a comprehension as a specific loop to make it clearer.

```
[2 * x if x % 2 == 0 else 3 * x for x
in [2, 3]]
```

Lambda and sorting

Q13. What will `my_list` contain after this code has been run?

```
my_list = ["abx", "cdk", "ghy"]
my_list.sort(key=lambda x: x[2])
```

Lambda and sorting

Q13. What will `my_list` contain after this code has been run?

```
my_list = ["abx", "cdk", "ghy"]
my_list.sort(key=lambda x: x[2])

# lambda x: x[2] access char in position 2
# key= sort by whatever func returns
# .sort sort in place
# "abx", "cdk", "ghy" -> k, x, y

print(my_list)
['cdk', 'abx', 'ghy']
```

Should be **sort** not **sorted**… error would be marked correct in practice questions!

Sort alphabetically by whatever the lambda function returns

Class vs instance attributes

Q14. What does this print?

```
class A:
    name = "My name"
    def __init__(self, s):
        self.name = s

a = A('Hugo')
print(a.name)
print(A.name)
```

Class vs instance attributes

Q14. What does this print?

```
class A:
    name = "My name"
    def __init__(self, s):
        self.name = s

a = A('Hugo')
print(a.name) # instance attribute -> “Hugo”
print(A.name) # class attribute -> “My name”

“Hugo”
“My name”
```

`A` is the class
`a` is an INSTANCE of the class

Class definition sets name as “My name” by default

Instantiation of an object of that class uses `__init__` and replaces name with argument

Capitalisation matters!

Instance attributes with kwargs

Q15. What does z evaluate to after this code is run?

```
class C:
    def __init__(self, x=1, y=2):
        self.x = x
        self.y = y
    def sum(self):
        return self.x + self.y

z = C(y=3).sum()
```

Instance attributes with kwargs

Q15. What does z evaluate to after this code is run?

```
class C:
    def __init__(self, x=1, y=2):
        self.x = x
        self.y = y
    def sum(self):
        return self.x + self.y

z = C(y=3).sum()
# z is instance of C
# we passed kwarg y=3 -> self.y = 3
# by default x=1 -> self.x = 1
# sum is 1 + 3
print(z)
4
```

Kwargs don’t have to be passed in order

Default values apply even with class inits

Inheritance

Q16. What does this print?

```
class A:
    def say(self):
        print("Hi there")

class B(A):
    def say(self):
        print("Howdy")
    def __init__(self):
        self.say()
        super().say()

A()
B()
```

Inheritance

Q16. What does this print?

```
class A:
    def say(self):
        print("Hi there")

class B(A):
    def say(self):
        print("Howdy")
    def __init__(self):
        self.say()
        super().say()

A() # -> just instantiate an object of class A
B() # -> instantiate uses own say and parent say
"Howdy"
"Hi there"
```

B inherits say method from A but overwrites with own say method definition

`super()` access parent attributes/methods

A has no special init so is just instantiated

B's init calls own say THEN parent say methods

Dictionaries and aliasing

Q17. What does `data[‘z’]` contain?

```
data = {'x': [1, 2, 3], 'y': [4, 5, 6]}
data['x'].append(4)
data['z'] = data['x']
data['x'][0] = 10
```

Dictionaries and aliasing

Q17. What does `data[‘z’]` contain?

```
data = {'x': [1, 2, 3], 'y': [4, 5, 6]}
data['x'].append(4)
# {'x': [1, 2, 3, 4], 'y': [4, 5, 6]}

data['z'] = data['x'] # ALIAS OF SAME LIST
# {'x': [1, 2, 3, 4], … 'z': [1, 2, 3, 4]}

data['x'][0] = 10
# {'x': [10, 2, 3, 4], … 'z': [10, 2, 3, 4]}

print(data[‘z’])
[10, 2, 3, 4]
```

Break down step by step:

- Add 4 to x’s list value
- Make ‘z’ refer to same list
- Modify this list

Modifying dictionaries

Q18. What does `my_dict` and `new_dict` contain?

```python
def modify_dict(d):
    d['a'] = 5
    d = {'a': 3, 'b': 4}
    return d

my_dict = {'a': 1, 'b': 2}
new_dict = modify_dict(my_dict)
```

Modifying dictionaries

Q18. What does `my_dict` and `new_dict` contain?

```
def modify_dict(d):
    d['a'] = 5
    d = {'a': 3, 'b': 4}
    return d

my_dict = {'a': 1, 'b': 2}
new_dict = modify_dict(my_dict)
# replace value of ‘a’ in d object - inplace change
# {‘a’: 5, ‘b’: 2}

# create new dict containing values
# {'a': 3, 'b': 4}

# return new_dict (but my_dict was still modified)

my_dict # {'a': 5, 'b': 2}
new_dict # {'a': 3, 'b': 4}
```

When passed `d` refers to the SAME diction as `my_dict`

Modifications to `my_dict` are side-effect of `modify_dict`

`d =` creates a new dict object and assigns to `d` name which is then returned explicitly and assigned to `new_dict`

Comprehension and dictionary iteration

Q19. What will this code print?

```
prices = {'apple': 0.5,
          'banana': 0.3,
          'orange': 0.6}
quantities = {'apple': 5,
              'orange': 3,
              'banana': 2}
print(sum(prices[fruit] * quantities[fruit] for fruit
in prices.keys()))
```

Comprehension and dictionary iteration

Q19. What will this code print?

```
prices = {'apple': 0.5,
          'banana': 0.3,
          'orange': 0.6}
quantities = {'apple': 5,
              'orange': 3,
              'banana': 2}
print(sum(prices[fruit] * quantities[fruit] for fruit
in prices.keys()))

new_list = []
for fruit in [‘apple’, ‘banana’, ‘orange’]:
     x = prices[fruit] * quantities[fruit]
     # 0.5 * 5 = 2.5
     # 0.3 * 2 = 0.6
     # 0.6 * 3 = 1.8
     new_list.append(x)
sum(new_list) # 2.5 + 0.6 + 1.8
4.9
```

Can convert to explicit loop if comprehension is hard to read

- .keys() iterates over keys of a dictionary

- dict[key] returns the value for that key

Docstrings

Q20. Write a docstring for this function

```python
def calculate_discount(price, percentage):
    """
    [Write your docstring here]
    """
    if not isinstance(price, (int, float)) or not isinstance(percentage, (int, float)):
        raise TypeError("Price and percentage must be numbers")
    if percentage < 0 or percentage > 100:
        raise ValueError("Percentage must be between 0 and 100")

    discount = price * (percentage / 100)
    return price - discount
```

Docstrings

Docstring should:

- Explain purpose of function
- Explain arguments and their requirements
- Explain what is returned

Q20. Write a docstring for this function

```python
def calculate_discount(price, percentage):
    """
    Calculate the final price after applying a discount percentage.

    Args:
      price (int or float): The original price
      percentage (int or float): The discount percentage between 0 and 100

    Returns:
      Price after applying the discount as a float
    """
    if not isinstance(price, (int, float)) or not isinstance(percentage, (int, float)):
        raise TypeError("Price and percentage must be numbers")
    if percentage < 0 or percentage > 100:
        raise ValueError("Percentage must be between 0 and 100")

    discount = price * (percentage / 100)
    return price - discount
```

Other concepts to solidify

- Correct syntax
    - Valid variable names
    - Valid ways of delineating numbers
- Gotchas with certain variable types
- Exceptions:
    - Try-Except-Else-Finally
    - Common types of exceptions
- Reading and writing files
    - `open` with different modes (‘r’, ‘w’, ‘a’)
    - Filehandles and their reading/writing methods
    - Endline characters
- Generators and Iterators
    - What yield does
    - What next does
    - Exhausting/partially using them up