National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 1
Introduction to CS1010X, Python & Functional Abstraction

Overview

1. Fun with IDLE
2. Assignment & Operators
3. Boolean operations
4. Conditional Statements
5. Functions

Python

1. *if - elif - else:*

   ```
   if expression:
       statement(s)
   elif expression:
       statement(s)
   else:
       statement(s)
   ```

   Consider each pre-condition *in sequence*, if the value of the any expression is not `False`, evaluate the corresponding statement(s). Otherwise evaluate the statement(s) under else.

2. Ternary Operator Form [To read only. Not expected to write code like that.]

   `[on_true] if [expression] else [on_false]`

   is equivalent to

   ```
   if [expression]:
       [on_true]
   else:
       [on_false]
   ```

Problems

1. Python supports a large number of different binary operators. Experiment with each of these, using arguments that are both integer, both floating point, and both

string. Not all operators work with each argument type. In the following table, put a cross in the appropriate boxes corresponding to the argument and operator combinations that result in error.

Operator	**Integer**	**Floating point**	**String**
`+`			
`-`			
`*`			
`/`			
`**`			
`//`			
`%`			
`<`			
`>`			
`<=`			
`>=`			
`==`			
`!=`			

Some of these operators were not discussed in lecture. Find out what they do. You might be asked to explain them to the rest of the class in recitation.

2. Evaluate the following expressions assuming `x` is bound to 3, `y` is bound to 5 and `z` is bound to -2:

```
x + y / z

x ** y % x

y <= z

x > z * y

y // x

x + z != z + x
```

```
if True:
    1 + 1
else:
    17

if False:
    False
else:
    42

if (x > 0):
    x
else:
    (-x)

if 0:
    1
else:
    2

if x:
    7
else:
    what-happened-here

if True:
    1
elif (y>1):
    False
else:
    wake-up
```

3. Suppose we're designing an point-of-sale and order-tracking system for a new burger joint. It is a small joint and it only sells 4 options for combos: Classic Single Combo (hamburger with one patty), Classic Double With Cheese Combo (2 patties), and Classic Triple with Cheese Combo (3 patties), Avant-Garde Quadruple with Guacamole Combo (4 patties). We shall encode these combos as 1, 2, 3, and 4 respectively. Each meal can be *biggie_sized* to acquire a larger box of fries and drink. A *biggie_sized* combo is represented by 5, 6, 7, and 8 respectively, for combos 1, 2, 3, and 4 respectively.

   (a) Write a function called `biggie_size` which when given a regular combo returns a *biggie_sized* version.

(b) Write a function called `unbiggie_size` which when given a *biggie_sized* combo returns a non-*biggie_sized* version.

(c) Write a function called `is_biggie_size` which when given a combo, returns `True` if the combo has been *biggie_sized* and `False` otherwise.

(d) Write a function called `combo_price` which takes a combo and returns the price of the combo. Each patty costs $1.17, and a *biggie_sized* version costs $.50 extra overall.

(e) An order is a collection of combos. We'll encode an order as each digit representing a combo. For example, the order 237 represents a Double, Triple, and *biggie_sized* Triple. Write a function called `empty_order` which takes no arguments and returns an empty order which is represented by 0.

(f) Write a function called `add_to_order` which takes an order and a combo and returns a new order which contains the contents of the old order and the new combo. For example, `add_to_order(1,2) -> 12`.

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 2
Recursion, Iteration & Orders of Growth

Definitions

Theta (Θ) notation:

$$f(n) = \Theta(g(n)) \;\Leftrightarrow\; \exists k_1, k_2, n_0\ .\ \ k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n), \text{for } n > n_0$$

Big-O notation:

$$f(n) = O(g(n)) \;\Leftrightarrow\; \exists k, n_0\ .\ \ f(n) \leq k \cdot g(n), \text{for } n > n_0$$

Adversarial approach: For you to show that $f(n) = \Theta(g(n))$, you pick k_1, k_2, and n_0, then I (the adversary) try to pick an n which doesn't satisfy $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$.

Terminology:

- $\exists$: There exists.
- $\Leftrightarrow$: If and only if (either both statements are true, or both are false).

Implications

Ignore constants. Ignore lower order terms. For a sum, take the larger term. For a product, multiply the two terms. Orders of growth are concerned with how the effort scales up as the size of the problem increases, rather than an exact measure of the cost.

Typical Orders of Growth

- $\Theta(1)$ - Constant growth. A fixed number of simple, non-decomposable operations have constant growth.
- $\Theta(\log n)$ - Logarithmic growth. At each iteration, the problem size is scaled down by a constant amount.
- $\Theta(n)$ - Linear growth. At each iteration, the problem size is decremented by a constant amount.
- $\Theta(n \log n)$ - Nifty growth. Nice recursive solution to normally $\Theta(n^2)$ problem.
- $\Theta(n^2)$ - Quadratic growth. Computing correspondence between a set of n things, or doing something of cost n to all n things both result in quadratic growth.
- $\Theta(2^n)$ - Exponential growth. Really bad. Searching all possibilities usually results in exponential growth.

What's n?

Order of growth is *always* in terms of the size of the problem. Without stating what the problem is, and what is considered primitive (what is being counted as a "unit of work" or "unit of space"), the order of growth doesn't have any meaning.

Problems

1. Remember our point-of-sale and order-tracking system from last week? Recall that the joint only sells 4 options for combos: Classic Single Combo (hamburger with one patty), Classic Double With Cheese Combo (2 patties), and Classic Triple with Cheese Combo (3 patties), Avant-Garde Quadruple with Guacamole Combo (4 patties). We shall encode these combos as 1, 2, 3, and 4 respectively. Each meal can be *biggie-sized* to acquire a larger box of fries and drink. A *biggie-sized* combo is represented by 5, 6, 7, and 8 respectively, for combos 1, 2, 3, and 4 respectively.

   In addition, an order is a collection of combos. We'll encode an order as each digit representing a combo. For example, the order 237 represents a Double, Triple, and *biggie-sized* Triple.

   Assume that you have the following functions available:

   - `biggie_size` which when given a regular combo returns a *biggie-sized* version.
   - `unbiggie_size` which when given a *biggie-sized* combo returns a non-*biggie-sized* version.
   - `is_biggie_size` which when given a combo, returns `True` if the combo has been *biggie-sized* and `False` otherwise.
   - `combo_price` which takes a combo and returns the price of the combo.
   - `empty_order` which takes no arguments and returns an empty order which is represented by 0.
   - `add_to_order` which takes an order and a combo and returns a new order which contains the contents of the old order and the new combo. For example, `add_to_order(1,2) -> 12`.

   (a) Write a recursive function called `order_size` which takes an order and returns the number of combos in the order. For example, `order_size(237) -> 3`.

   (b) Write an iterative version of `order_size`.

(c) Write a recursive function called `order_cost` which takes an order and returns the total cost of all the combos.

(d) Write an iterative version of `order_cost`.

(e) **Homework:** Write a function called `add_orders` which takes two orders and returns a new order that is the combination of the two. For example, `add_orders (123,234) -> 123234`. Note that the order of the combos in the new order is not important as long as the new order contains the correct combos. `add_orders(123,234) -> 122334` would also be acceptable.

2. Give order notation for the following:

   (a) $5n^2 + n$

   (b) $\sqrt{n} + n$

   (c) $3^n n^2$

3.
```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

   Running time? Space?

4. Write an iterative version of `fact`.

5.
```
def find_e(n):
    if n == 0:
        return 1
    else:
        return 1/fact(n) + find_e(n - 1)
```

   Running time? Space? (Assume iterative `fact`)

6. Assume you have a function `is_divisible(n, x)` which returns `True` if `n` is divisible by `x`. It runs in $O(n)$ time and $O(1)$ space. Write a function `is_prime` which takes a number and returns `True` if it's prime and `False` otherwise.

   Running time? Space?

7. **Homework:** Write an iterative version of `find_e`.

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 3
Higher Order Functions

Python

1. *lambda* - `lambda` *params:* `expression`
   Creates an anonymous function equivalent to:

   ```
   def function(params):
       return expression
   ```

Definitions

The following are two higher-order functions discussed in lecture:

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def fold(op, f, n):
  if n == 0:
    return f(0)
  else:
    return op(f(n), fold(op, f, n-1))
```

Note: it is not necessary to memorize these defintions, or even the names of these functions. Definitions of such functions (if they are used) will be given in an Appendix for examinations. What you need to be able to do is to read the definition for such functions and understand what they do and be able to use them.

Problems

1. Evaluate the return values of the following expressions:

   (a)
   ```
   x = 2
   def f():
       x = 5
       y = x + 5
       return x + y
   f() + x
   ```

(b)
```
x = 2
y = 3
def g(x):
    y = x + 5
    x = 7
    return x + y
g(y)+y
```

(c)
```
x = 4
def foo(x):
    return x(3)
foo(lambda x: x+1)
```

(d)
```
x = 5
def bar(x, y):
    return y(x)
bar(4, lambda x: x**2)
```

2. Write a function `my_sum` that computes the following sum, for $n \geq 1$:

$$1 \times 2 + 2 \times 3 + \cdots + n \times (n + 1)$$

3. Is the function `my_sum` as defined above a recursive process or an iterative process? What is the order of growth in time and in space?

4. If your answer in Question 2 is a recursive process, re-write `my_sum` as an iterative process. If your answer in Question 2 is an iterative process, re-write `my_sum` as a recursive process. What is the new order of growth in time and space?

5. We can also define `my_sum` in terms of the higher-order function `sum`. Complete the definition of `my_sum` below. You cannot change the definition of `sum`; you may only call it with appropriate arguments.

```
def my_sum(n):
    return sum(<T1>, <T2>, <T3>, <T4>)
```

```
T1:
```

```
T2:
```

```
T3:
```

```
T4:
```

6. Suppose instead we define `my_sum` in terms of the higher-order function `fold`. Complete the definition of `my_sum` below.

```
def my_sum(n):
    return fold(<T1>, <T2>, <T3>)
```

```
T1:
```

```
T2:
```

```
T3:
```

7. Write an iterative version of `sum`.

8. **Homework:** Write an iterative version of `fold`.

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 4
Data Abstraction

Python

1. *Tuple* - (*value1, value2, ...*)

   A tuple is an immutable sequence of Python objects enclosed in parentheses and separated by commas.

2. Operations on tuples:
   (a) *len(x)* - Returns the number of elements of tuple x.
   (b) *element* in *x* - Returns `True` is *element* is in x, and `False` otherwise.
   (c) for *var* in *x* - Will iterate over all the elements of x with variable var.
   (d) *max(x)* - Returns the maximum element in the tuple x.
   (e) *min(x)* - Returns the miniimum element in the tuple x.

Problems

1. Evaluate the following expressions:

```
tup_a = (10, 12, 13, 14) #Creating tup_a
print(tup_a)

tup_b = ("CS1010X", "CS1231") #Creating tup_b
print(tup_b)

tup_c = tup_a + tup_b #Creating tup_c
print(tup_c)

len(tup_c)

14 in tup_a

11 in tup_c

tup_d = tup_b[0] * 4

tup_d[0]

tup_d[1:]
```

```
count = 0
for i in tup_a:
    count = count + i
print(count)

max(tup_a)

min(tup_a)

max(tup_c)

min(tup_c)
```

2. Write expressions whose values will print out like the following.

```
(1, 2, 3)
```

```
(1, (2), 3)
```

```
(1, (2,), 3)
```

```
((1, 2), (3, 4), (5, 6))
```

3. Write expressions to that will return the value 4 when the `x` is bound to the following values:

```
(7, 6, 5, 4, 3, 2, 1)
```

```
(7, (6, 5, 4), (3, 2), 1)
```

```
(7, ((6, 5, (4,), 3), 2), 1)
```

4. You found a holiday assignment at the Registar's Office. Your job is to write a program to help students with their scheduling of classes. You are provided with an implementation of the records for each class as follows:

```
def make_module(course_code, units):
    return (course_code, units)

def make_units(lecture, tutorial, lab, homework, prep):
    return (lecture, tutorial, lab, homework, prep)

def get_module_code(course):
    return course[0]

def get_module_units(course):
    return course[1]

def get_module_total_units(units):
    return units[0] + units[1] + units[2] + units[3] + units[4]
```

Each class (course) has a course code and an associated number of credit unit, e.g. for CS1101S, that's 3-2-1-3-3. Your job is now to write a schedule object to represent the sets of classes taken by a student. **Note:** Since `class` is a keyword in Python, we will use `course` as the variable representing the current class of interest.

(a) Write a constructor `make_empty_schedule()` that returns an empty schedule.

```
def make_empty_schedule():
```

Order of growth in time, space?

(b) Write a function `add_class` that when given a class and a schedule, returns a new schedule including the new class:

```
def add_class(course, schedule):
```

Order of growth in time, space?

(c) Write a function `total_scheduled_units` that computes the total number of units in a specified schedule.

```
def total_scheduled_units(schedule):
```

Order of growth in time, space?

(d) Write a function `drop_class` that returns a new schedule with a particular class dropped from a specified schedule.

```
def drop_class(schedule, course):
```

Order of growth in time, space?

(e) Implement a credit limit by taking in a schedule, and returning a new schedule that has total number of units is less than or equal to `max_credits` by removing classes from the specified schedule.

```
def credit_limit(schedule, max_credits):
```

Order of growth in time, space?

(f) **Homework:** Implement an improved version of `credit_limit` that will return a schedule with a total number of units is less than or equal to `max_credits`, but with the maximal number of classes. What is the order of growth of your solution? Is that the best you can do?

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 5
Working with Sequences

Python

1. Equality Testing
   - `==` returns `True` if two objects are equivalent.
   - `is` returns `True` if two objects are identical, i.e. they are the same object.
2. Membership Testing
   - x `in` y returns `True` if x is contained in the sequence type y. There are three basic sequence types in Python: tuples, lists, and range objects. We will discuss lists after the midterm break.
3. Range Type
   - The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.
   - `range(start, stop[, step])`: Creates a sequence starting at `start`, at intervals of `step`, up to (but not including `stop`). If `step` is zero, `ValueError` will be raised.
   - `range(stop)` : Creates a sequence starting at 0, at intervals of 1, up to (but not including `stop`).
   - The advantage of the range type over a regular tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed), i.e. $O(1)$ space.

Problems

1. Evaluate the following expressions:

```
1 == 1

1 is 1

"foo" == 'foo'

0 == "0"

0 is "0"
```

```
False == False

False == 0

(1, 2) == (1, "2")

(1, 2) is (1, 2)

(1, 2, 3, 4, 5) == (1, 2, 3, 4, 5)

(1, 2, 3, 4, 5) == (1, 2, 3, 5, 4)

((1, 2), (2, 3)) == ((1, 2), (3, 2))

x = (1,2)
y = (1,2)
z = x

x is y

x == y

x is z

3 in (1, 2, 3, 4, 5)

(1, 2) in (1, 2, 3, 4, 5)

(2) in (1, 2, 3, 4, 5)

() == ()

(1) == 1

(1, ) == 1

a = ((1,2), (3,4))

b = (x, (3,4))

x in a

x in b
```

2. The function `in` will test if an object is inside a tuple by checking for equivalence. Write a function `contains` that will check if an object is inside a tuple by checking for identity. For example,

```
x = (1,2)
a = ((1,2), (3,4))
b = (x, (3,4))
```

```
contains(x,a) => False
contains(x,b) => True
```

Write a function `deep_contains` that will check if an object is nested arbitrarily deep within a tuple. For example,

```
x = (1,2)
a = ((1,2), ((3,4), x), (5,6))

contains(x,a) => False
deep_contains(x,a) => True
```

3. The `accumulate` procedure discussed in lecture is also known as `fold_right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold_left`, which is similar to `fold_right`, except that it combines elements working in the opposite direction:

```
def accumulate(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(seq[0], accumulate(fn, initial, seq[1:]))

def fold_left(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(fold_left(fn, initial, seq[:-1]),seq[-1])
```

   (a) Suppose we also define the following functions:

```
def pair(a, b):
    return (a, b)

def divide(a, b):
    return a/b
```

   What are the values of

```
fold_right(divide, 1,(1, 2, 3))
fold_left(divide,1,(1, 2, 3))
fold_right(pair,(),(1, 2, 3))
fold_left(pair,(),(1, 2, 3))
```

(b) Give a property that `fn` should satisfy to guarantee that `fold_right` (or `accumulate`) and `fold_left` will produce the same values for any sequence.

4. A *queue* is a data structure that stores elements in order. Elements are enqueued onto the tail of the queue. Elements are dequeued from the head of the queue. Thus, the first element enqueued is also the first element dequeued (FIFO, first-in-first-out). The `qhead` operation is used to get the element at the head of the queue.

```
qhead(enqueue(5, empty_queue()))
# Value: 5

q = enqueue(4, enqueue(5, enqueue(6, empty_queue())))

qhead(q)
# Value: 6

qhead(dequeue(q))
# Value: 5
```

(a) Decide on an implementation for *queue*.

(b) Implement `empty_queue`

```
def empty_queue():
```

Order of growth in time? Space?

(c) Implement `enqueue`; a function that returns a new queue with the element `x` added to the tail of `q`.

```
def enqueue(x, q):
```

Order of growth in time? Space?

(d) Implement `dequeue`; a function that returns a new queue with the head element removed from `q`.

```
def dequeue(q):
```

Order of growth in time? Space?

(e) Implement `qhead`; a function that returns the value of the head element of `q`.

```python
def qhead(q):
```

Order of growth in time? Space?

5. **Homework:** Suppose `x` is bound to the tuple `(1, 2, 3, 4, 5, 6, 7)`. Using `map`, `filter`, `accumulate` and/or lambdas (as discussed in Lecture), write an expression involving `x` that returns:

   (a) `(1, 4, 9, 16, 25, 36, 49)`

   (b) `(1, 3, 5, 7)`

   (c) `((1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7))`

   (d) `((2, 4), 6)`

   (e) `(((2,), 4), 6)`

   (f) The maximum element of `x`: `7`

   (g) The minimum element of `x`: `1`

   (h) The maximum squared even element of `x`: `36`

   (i) The sum of the square of each value in `x`: `140`

   You are encouraged to provide multiple solutions for the above questions.

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 6
Lists, Searching & Sorting

Python

1. *[]* - list constructor
   By itself, creates an empty list. Initialize with elements in this manner:
   `[x1, x2, ..., xN]`
2. *list* - `list(<`*sequence*`>)`
   Takes in a *sequence* type and converts it into a list. If no sequence is provided, i.e. `list()`, an empty list is returned.
3. Common Sequence Operations (works on lists, tuples and strings):
   (a) *len(x)* - Returns the number of elements of list x.
   (b) *element* in *x* - Returns `True` is *element* is in x, and `False` otherwise.
   (c) for *var* in *x* - Will iterate over all the elements of x with variable var.
   (d) *max(x)* - Returns the maximum element in the list x.
   (e) *min(x)* - Returns the minimum element in the list x.
4. Mutation Operations for list *lst*:
   (a) *lst.append(x)* - Modifies list by adding an element x.
   (b) *lst.extend(x)* - Modifies list by adding a another list x.
   (c) *lst.copy()* - Returns a shallow copy of *lst*.
   (d) *lst.reverse()* - Modifies *lst* by reversing it.
   (e) *lst.insert(i, x)* - Inserts element x at index i.
   (f) *lst.pop()* - Removes the last element of *lst* and returns it.
   (g) *lst.pop(i)* - Removes the element at index i in *lst* and returns it.
   (h) *lst.remove(x)* - Modifies list by removing the first occurence of the element x.
   (i) *lst.clear()* - Empties the list *lst*.

Problems

1. Evaluate the following expressions:

```python
many_things = [1, 'a', ('I', 'can', 'have', 'tuples', 'in', 'lists')]
print(many_things)

numbers = [2, 3, 4]
print(numbers)

concatenated = many_things + numbers
print(concatenated)

appended = many_things.append(numbers)
print(appended)
print(many_things)

extended = many_things.extend(numbers)
print(extended)
print(many_things)

many_things[0] = 7
print(many_things)

can_be_indexed = concatenated[2]
print(can_be_indexed)

can_be_indexed_multiple_times = concatenated[2][1]
print(can_be_indexed_multiple_times)

a_shallow_copy = concatenated[:]
print(a_shallow_copy)
print(a_shallow_copy == concatenated)
print(a_shallow_copy is concatenated)

woops = a_shallow_copy[2]
print(woops)
print(woops is can_be_indexed)

singleton = ['blah']
print(singleton)
```

2. In lecture, we implemented selection and merge sort, but we did so without using indices. The "typical" way to implement sorts is to use list indices. Implement the following sort:

   (Bubble Sort) Start at index 0, iterate down the list exchanging two pairs of elements if the second is smaller than the first. Rinse and repeat until there are no swaps required during a given pass.

   What is the space and time complexity for your implementation?

3. **(In-place Selection Sort)** Now, implement selection sort, which was described in class, as an in-place sort by using list indices, i.e. write a function `selection_sort` that takes as its argument an unsorted list and modifies it so that it is sorted according to the comparison operator $<$.

   What is the space and time complexity for your implementation?

4. Given the following list:

```
students = [
    ('tiffany', 'A', 15),
    ('jane', 'B', 10),
    ('ben', 'C', 8),
    ('simon', 'A', 21),
    ('john', 'A', 15),
    ('jimmy', 'F', 1),
    ('charles', 'C', 9),
    ('freddy', 'D', 4),
    ('dave', 'B', 12)]
```

   (a) How would you sort the list of students by name in reverse alphabetical order?

   (b) How would you sort the list of students by letter grade, followed by name in alphabetical order?

   (c) How would you obtain a tuple of all the names with fewer than 6 characters?

   (d) How would you obtain a tuple of pairs, where the first element is a letter grade and the second is the number of occurrences of that letter grade?

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 7
Multiple Representations

Problems

1. **Dense Matrix Representation**. A matrix can be represented in Python by a list of lists (nested lists). For example, `m = [[ 1, 2, 3], [4, 5, 6], [7, 8, 9]]` represents the following 3×3 matrix:

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

You are given the following implementation for `make_matrix(seq)`, which takes in a sequence, i.e. either a tuple or a list, and creates the matrix object.

```python
def make_matrix(seq):
    mat = []
    for row in seq:
        mat.append(list(row))
    return mat
```

(a) Suppose `seq` were a list of lists. Would the following implementation of `make_matrix(seq)` work? Explain.

```python
def make_matrix(seq):
    return seq
```

(b) Implement the following supporting functions:

i. `rows(m)`: returns the number of rows for matrix object `m`.

ii. `cols(m)`: returns the number of columns for matrix object `m`.

iii. `get(m,i,j)`: returns the element (i,j) for matrix object `m`.

iv. `set(mat,i,j,val)`: sets the element (i,j) for matrix object `m` to value `val`.

v. `transpose(m)`: transposes matrix object `m`. Basically, this converts a $m \times n$ matrix into a $n \times m$ matrix.

vi. `print_matrix(mat)`: prints the contents of matrix object `m` in a human readable form.

2. **Sparse Matrix Representation**. Now suppose that implementation of `make_matrix(seq)` is as follows:

```
def make_matrix(seq):
    data = []
    for i in range(len(seq)):
        for j in range(len(seq[0])):
            if seq[i][j] != 0:
                data.append([i,j,seq[i][j]])
    return [len(seq),len(seq[0]),data]
```

   (a) Implement the list of associated functions listed in Part 1(ii) above.

      i. `rows(m)`

      ii. `cols(m)`

      iii. `get(m,x,y)`

      iv. `set(mat,x,y,val)`

v. `transpose(m)`

vi. `print_matrix(mat)`

(b) Which is the better implementation for the matrix object? Explain.

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 8
Dictionaries & Message Passing

Python

1. *{}* - dictionary constructor

   By itself, creates an empty dictionary. Initialize with elements in this manner: `{key1:element1, key2:element2, ···, keyN:elementN}`

2. *dict* - `dict(<`*sequence of pairs*`>)`

   Takes in a *sequence* type consisting of sequence of pairs (tuples) and converts it into a dictionary. If no sequence is provided, i.e. `dict()`, an empty dictionary is returned. If the provided sequence is not a sequence of pairs, this will cause an error.

3. Assignment - `<dict>[`*key*`] = ` *value*. Assigns a new *value* to the specified *key* in the dictionary `<dict>`. This updates an existing record if one exists, and creates a new record if none exists.

4. Deletion:

   (i) `del <dict>[`*key*`]`. Deletes the record corresponding to the specified *key* in the dictionary `<dict>`, if one exists.

   (ii) `<dict>.clear()`. Remove all entries in `<dict>`.

   (iii) `del <dict>`. Deletes the dictionary `<dict>`.

5. Access:

   (i) `<dict>.get(key, default=None)`. For key *key*, returns value, or default if *key* is not in dictionary `<dict>`.

   (ii) `key in <dict>`. Returns `True` if *key* in dictionary `<dict>`, `False` otherwise.

   (iii) `<dict>.keys()`. Returns list of dictionary `<dict>`'s keys.

   (iv) `<dict>.values()`. Returns list of dictionary `<dict>`'s values.

   (v) `<dict>.items()`. Returns a list of `<dict>`'s (key, value) tuple pairs

   (vi) `len(<dict>)`. Returns the number of elements in `<dict>`.

Problems

1. Evaluate the following expressions:

```
a = (("apple", 2), ("orange", 4), (5, 7))
b = dict(a)

c = [[1, 2], [3, 4], [5, 7]]
d = dict(c)

print(b["orange"])

print(b[5])

print(b[1])

b["bad"] = "better"
b[1] = "good"

for key in b.keys():
    print(key)

for val in b.values():
    print(val)

del b["bad"]
del b["apple"]

print(tuple(b.keys()))

print(list(b.values()))
```

2. **Stack Implementation (in Message-Passing Style).** Implement a stack object with the following functions:

   (i) `make_stack`: returns a new empty stack object.

   (ii) `s("is_empty")`: returns `True` if the stack `s` is empty.

   (iii) `s("clear")` : empties the stack `s` of any elements it may contain.

   (iv) `s("peek")`: returns the top element of the stack `s`, leaving the stack unchanged. If the stack is empty, returns `None`.

   (v) `s("push")(item)`: pushes an element `item` onto the top of the stack `s`.

   (vi) `s("pop")`: removes and returns the top element of the stack `s`. If the stack is empty, returns `None`.

Sample execution:

```
s = make_stack()
print(s("is_empty")) # True
s("push")(1)
s("push")(2)
print(s("peek"))     # 2
print(str(s("pop"))) # 2
print(str(s("pop"))) # 1
print(str(s("pop"))) # None
```

3. Write a function called `push_all` which takes a stack and a sequence and pushes all the elements of the sequence onto the stack. It should return the stack.

4. Write a function called `pop_all` which takes a stack and pops elements off it until it becomes empty, adding each element to an output list.

5. **Calculator Object Implementation**

```
def make_calculator():  #an RPN calculator
    stack = make_stack()
    ops = {'+':lambda x, y: x + y,
           '-':lambda x, y: x - y,
           '*':lambda x, y: x * y,
           '/':lambda x, y: x / y}
    def oplookup(msg, *args):
        # YOUR CODE BEGINS HERE


        # YOUR CODE ENDS HERE
        else:
            raise Exception("calculator doesn't" + msg)
    return oplookup


c = make_calculator()
print(c('ANSWER'))                # empty_stack
print(c('NUMBER_INPUT',4))        # pushed
print(c('ANSWER'))                # 4
print(c('NUMBER_INPUT',5))        # pushed
print(c('ANSWER'))                # 5
print(c('OPERATION_INPUT','+')) # pushed
print(c('ANSWER'))                # 9
print(c('NUMBER_INPUT',7))        # pushed
print(c('OPERATION_INPUT','-')) # pushed
print(c('ANSWER'))                # 2
print(c('CLEAR'))                 # cleared
print(c('ANSWER'))                # empty_stack
```

(i) Complete the definition of `oplookup` so it is a function that when given an operation name and the `ops` list, will return the operation with the given name.

(ii) Write a method called `ANSWER`, which returns the current value on the top of the stack.

(iii) Write a method called `CLEAR`, which removes all the numbers from the stack.

(iv) Write a method called `NUMBER_INPUT`, which puts the number onto the stack.

(v) Write a method called `OPERATION_INPUT`, which takes an operation name as input, looks up the operation, removes two numbers from the stack, and puts the result of the operation back onto the stack.

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 9
Object-Oriented Programming

Problems

1. Write a `Food` class
   - Input properties is the `name`, `nutrition` value, and `good_until` time.
   - Additional property is the `age` of the food, initially 0.
   - Methods are:
     - `sit_there` - takes an amount of time, and increases the age of the food by the amount.
     - `eat` - return the nutrition if the food is still good; 0 otherwise.

2. Write an `AgedFood` class
   - Input property is the same as the `Food` class, with an additional property, which is the `good_after` time.
   - Should inherit from the `Food` class.
   - Methods are:
     - `sniff` - returns `True` if it has aged enough to be good, `False` otherwise.
     - `eat` - returns 0 if the food is not good yet; otherwise behaves like normal food.

3. Write a `VendingMachine` class
   - Input property is the same as the `Food` class.
   - Additional property is `age` of the `VendingMachine`, initially 0.
   - Methods are:
     - `sit_there` - takes an amount of time, and increases the age of the vending-machine by *half* that amount (it's refridgerated!).
     - `sell_food` - returns a new food instance with the appropriate name, nutrition and good_until.

4. Write `mapn`, which allows an arbitrary number of input lists[1], for example:

```
mapn(lambda x,y,z: (z, x+y),
    ((1, 2, 3), (4, 5, 6), ('first', 'second', 'third')))

#Output:        (('first', 5), ('second', 7), ('third', 9))
```

   The function definition should look like this:

```
def mapn(fn, lsts)
# fn is the function that you would apply to the lsts
# lsts is the list of lists that would given as input to the function fn
```

   You may use the regular `map` in your implementation.

5. **Homework:** How would you implement the vending machine so that it can sell both Food and AgedFood (and possibly other things too?).

[1]It turns out that the regular `map` is pretty similar to the `mapn` you will write here!

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 10
Dynamic Programming & Memoization

Python

1. *Exception*

   Exceptions are error detected during execution. For example, `ZeroDivisionError`, `NameError` and `TypeError`

2. *Exception handling syntax*

```
try:
    statements
except <ErrorType>:
    statements
except (<ErrorType1>,<ErrorType2>, <ErrorType3>):
    statements
except:  # wildcard
    statements
finally: # always executed
    statements
```

3. *Optional else syntax*

```
try:
    statements
except <ErrorType>:
    statements
else:
    statements
```

4. Raising Exceptions

```
raise <ErrorType>
```

Problems

1. Recall the `count_change` algorithm shown in the earlier lectures. In a call to `cc(11, 5)`, there are actually computations of repeated subproblems, for example, repeated computation of `cc(1, 2)`.

```
# A certain path in the recursion tree:
cc(11, 5) -> cc(11, 4) -> cc(11, 3) -> cc(11, 2) -> cc(6, 2) -> cc(1, 2)
# Another path in the recursion tree:
cc(11, 5) -> cc(11, 4) -> cc(11, 3) -> cc(1, 3) -> cc(1, 2)
```

(a) Implement a memoized version of `cc(amount, kind_of_coins)` that uses a table to store previously calculated values.

(b) Implement a dynamic programming version of `cc(amount, kind_of_coins)` that fills up a table systematically.

2. Suppose you have a rod of length n meters and you hope to cut it into pieces and sell them. You can only cut them into integer lengths. The profit of rod with different lengths are displayed in the table below. How would you cut the rod to maximize the profit?

length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

Here is an example, suppose your rod is 4 meters. The different ways to cut rod and the corresponding profits are displayed below.

Do not cut at all. You make 9 dollars. Cut into 1 meter and 3 meters. You make $1 + 8 = 9$ dollars.
Cut into 2 meters and 2 meters. You make $5 + 5 = 10$ dollars. This is the optimal.
Cut into 1 meter, 1 meter and 2 meters. You make $1 + 1 + 5 = 7$ dollars.
Cut into four 1 meter rods. You make $1 + 1 + 1 + 1 = 4$ dollars.

Now write a function `cut_rod` that takes in an integer `n`, the length of the rod, and a dictionary mapping lengths to prices and returns the maximum profit that can be made. For example,

```
>>> prices = {1:1, 2:5, 3:8, 4:9, 5:10, 6:17, 7:17, 8:20, 9:24, 10:30}
>>> cut_rod(4, prices)
10
```

(a) Can you come up with a recursive solution?

(b) Can you give a solution that makes use of dynamic programming?

National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2022/2023

Recitation 11
Java Programming

Java Syntax

1. `switch` – Similar to Python `if/elif/elif/elif/.../else` conditional
2. Logical Operators `&&`, `||` and `!` – Equivalent to Python `and`, `or` and `not`
3. `for` (*initial, condition, update*) – `for` loop.
4. `Scanner#nextLine` – Getting user input
5. Returning multiple values
   - Use arrays.
   - Use Object Classes.

Problems

1. **Fibonacci Numbers.** Recall the Fibonacci numbers.
   (a) Define the function `int fib(int n)` that computes the n-th Fibonacci number iteratively and returns it.
   (b) Write a Java program that prints the 46th Fibonacci number. Compile your code and run your program.
   (c) Modify your program to print the 47th Fibonacci number. Is your printout correct? If not, explain why.

2. **Counting Change.** Write a Java program that prompts the user for an amount (in cents) and returns the number of ways to form that amount using 1¢, 5¢, 10¢, 20¢ and 50¢ coins.

3. **Returning Multiple Values.** Write a function that takes in (at least) an integer array and an integer n, and splits the array into two arrays—the first containing all the integers smaller than or equal to n and the second containing those larger.

   You may decide on the arguments and return values (if any) of the function. Write a Java program to illustrate the use of your function. State any assumptions you make.

   How can we create a sorting algorithm out of this function?