# Programming for Artificial Intelligence (Python)
# Homework 3
# **Due: March 29 before class**

We have six required and two optional questions in this homework. The first one is an analysis question, you can type your answer in a cell by commenting it:

```
# I think the logic is ...
```

The other questions are coding questions.

## 1 Question 1 (answer in words)

We frequently need to understand how functions or loops work in Python. One way to do so is to write out the loop manually. For example, below we define a `sum1` function:

```
def sum1(list):
    s = 0
    for i in list:
        s += i
    return s
sum1([1, 2, 3])
```

The logic evaluation of the function is

1. start `sum1([1,2,3])`: s = 0
2. i = 1, s = 1

3. i = 2, s = 3

4. i = 3, s = 6

5. return: 6

Write out the logic evaluation of the following function. Note: you need to keep track of **all variables**.

```
    def naive_by2(lst, by):
        n = len(lst) // by
        res = []
        for i in range(n):
            res.append([lst[2 * i], lst[2 * i+1]])
        return res
    nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    naive_by2(nums, 2)
```

# 2 Question 2 (code)

In class, we created a grouper to split `nums=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` into `[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]`. However, we wanted to split `nums` into `[[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]`. Modify the following function to achieve this goal. You do not need to use `iterators` or `generators`.

```
    def naive_by2(lst, by):
        n = len(lst) // by
        res = []
        for i in range(n):
            res.append([lst[2 * i], lst[2 * i+1]])
        return res
```

# 3 Question 3 (code)

Write a generator that gives out squared numbers like 1, 4, 9, 16, ... in two ways: the function way and the comprehension (generator expression) way. For the comprehension, you can stop at 100.

# 4 Question 4 (code)

Write a Python generator function that generates Fibonacci numbers indefinitely. The Fibonacci sequence is defined by the recurrence relation: $F(n) = F(n-1) + F(n-2)$, with initial values $F(0) = 0$ and $F(1) = 1$. Test your generator by printing the first 10 Fibonacci numbers. **Hint: you can try the function way**:

```
def fibonacci():
    pass
    while True:
        pass
```

# 5 Question 5 (code)

You need to write a generator expression to answer this question.

A generator expression is enclosed by parentheses. For example:

```
generator2 = (i for i in range(10))
```

**Note this is different from list comprehension which uses square brackets `[` and `]`.**

For a list of words, write a generator expression that generates a generator containing the lengths of words that start with a consonant (any letter except 'a', 'e', 'i', 'o', 'u') and end with a vowel ('a', 'e', 'i', 'o', 'u') from the given list.

For example, if the input list is `["apple", "banana", "orange", "kiwi", "grape"]`, the output generator should produce 6, 4, and 5, since "banana", "kiwi", and "grape" meet the criteria.

Test your program with the input list `["apple", "banana", "orange", "kiwi", "grape", "elephant", "tiger", "lion", "mouse"]`. **(Hint: you can use the `___contains___()` and the `len()` function)**

# 6 Question 6 (code)

This question needs you to create a `namedtuple`:

```
from collections import namedtuple
...
```

Suppose you are managing a database of students, and each student record consists of the following fields: "name", "age", "grade", and "major". Make a `namedtuple` type to store these information. Here are the possible steps:

1. Create a namedtuple called "Student" with fields "name", "age", "grade", and "major".
2. Create a list of Student records representing the following students:
   - Alice, 20 years old, grade A, major Economics
   - Bob, 22 years old, grade B, major Mathematics
   - Charlie, 21 years old, grade B, major Physics
3. Print the details of all students in the list.

# 7 Question 7 (Cross Validation, Optional)

In our second theory class on Friday, we learned using cross validation to choose (or more formally, tune) hyperparameters. This exercise helps us practice hyperparameter tuning manually. Consider again the data

| i | y | x |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 0 | 1 |
| 3 | 1 | 2 |
| 4 | 3 | 3 |
| 5 | 5 | 4 |

To cross validate. Let's consider split the data into five folds:

| i | y | x | fold |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 2 |
| 3 | 1 | 2 | 3 |
| 4 | 3 | 3 | 4 |
| 5 | 5 | 4 | 5 |

The model we are going to validate is $y = f(x; a, b) = a + bx + \varepsilon$.
Here is how cross validation works:

1. Step 1. Use folds 1,2,3,4 to estimate the parameters:

$$\min \sum_{i=1,2,3,4} \left(y^{[i]} - a - bx^{[i]}\right)^2$$

   Denote the estimates as $\hat{a}_{1:4}$ and $\hat{b}_{1:4}$. (**Hint: you can write a function to facilitate computation of $s_{xy}$ and $s_{xx}$.**)

2. Step 2. Apply $\hat{a}_{1:4}$ and $\hat{b}_{1:4}$ to fold 5: $\hat{y}^{[5]} = \hat{a}_{1:4} + \hat{b}_{1:4}x^{[5]}$.

3. Step 3. Find the squared residual for fold 5: $\left(r^{[5]}\right)^2 = \left(y^{[5]} - \hat{y}^{[5]}\right)^2$.

4. Step 4. Repeat Steps 1-3 but switch folds:

   (a) use 1,2,3,5 to estimate $\hat{a}_{1:3,5}$ and $\hat{b}_{1:3,5}$ and apply to fold 4. Then compute $\left(r^{[4]}\right)^2$.

   (b) use 1,2,4,5 to estimate $\hat{a}_{1:2,4:5}$ and $\hat{b}_{1:2,4:5}$ and apply to fold 3. Then compute $\left(r^{[3]}\right)^2$.

   (c) use 1,3,4,5 to estimate $\hat{a}_{1,3:5}$ and $\hat{b}_{1,3:5}$ and apply to fold 2. Then compute $\left(r^{[2]}\right)^2$.

   (d) use 2,3,4,5 to estimate $\hat{a}_{2:5}$ and $\hat{b}_{2:5}$ and apply to fold 1. Then compute $\left(r^{[1]}\right)^2$.

5. Step 5. Add up $\left(r^{[i]}\right)^2$ to have a validation loss for the model $f(x; a, b)$. Denote it as $v_1 = \sum_{i=1}^{5} \left(r^{[i]}\right)^2$.

**The following is only for illustration. You don't need to write code.**

$v_1$ gives the *validation loss* of the model $f(x; a, b) = a + bx$. The next step is to validate another model. You need to change the model to $f(x; a, b_1, b_2) = a + b_1x + b_2x^2 + \varepsilon$ and repeat the prior steps. Denote the validation loss as $v_2$.

If $v_2 > v_1$, $f(x; a, b)$ is a better model than $f(x; a, b_1, b_2)$ in terms of *out-of-sample* performance.

Suppose you do the same procedure for aother models. For example, $f(x; a, b_1, b_2, b_3) = a + b_1x + b_2x^2 + b_3x^3 + \varepsilon$. You van compare $v_1$, $v_2$, and $v_3$ to decide which model to use for out-of-sample *prediction*.

# 8 Question 8 (The `iter` function, Optional)

There are four pieces of information to help you answer this question (but you can choose to use a subset of them):

1. One advantage of iterator (or more specifically, generator) is that it can have unlimited number of elements. For example, if you **were** to use the functional way, you **could** have done:

```
def infinity_gen():
  natural1 = 0
  while True:
    yield natural1
    natural1 += 1
```

   This gives you a generator which produces **all** natrual numbers.

2. In Python, you can make a random number with the `random` library:

```
import random
r1 = random.choice(range(10))
```

   Then `r1` is a random number chosen from integers `{0,1,2, ... ,9}`.

3. Global variable. When you define a function in Python, you can define a variable from within the function:

```
def func(x):
  a = x + 1
  return a
```

   The variable `a` resolves after the function `func` is called and run. However, if you want the function to change the values outside of its definition, you can use the `global` keyword:

```
var1 = 1
def func(x):
  global var1
  var1 = x + 1
func(2)
print(var1)
```

4. In class, we talked about that we can turn an **iterable** into an **iterator**:

```
l1 = [1,2,3,4,5]
i1 = iter(l1)
```

But `iter` is more powerful. **Another** use of the `iter` function is that you can pass two arguments to it: `iter(f, s)`. Here, `f` is a function and `s` is the value at which the iterator stops.

**Hint: This means you call `f` iteratively until the `return` value of `f` equals `s`.**

For example:

```
var2 = 0
def f():
    global var2
    var2 = var2 + 1
    return var2
i2 = iter(f, 3)
```

Then `i2` will have two elements when you call it: `1,2`.

This is because when you run `iter(f, 3)`, Python calls `f` once and `var2=var2+1` is returned. The global `var2` is `1`. Then `iter` calls the `f` function again and `var2=var2+1` is returned for the second time. But `var2` is `2` now. Next, for the third time, `iter` calls `f` and `var2=var2+1` is `3`, which equals `s` and the iteration ends.

In this question, you will need to iteratively make random numbers from {0,1,2,3,4,5,6,7,8,9} until it gives 0 for the first time. Print out all the random numbers before the `0`. For example, if you use a `while` loop, the code is:

```
import random
i = random.choice(range(10))
while i ! = 0:
  print(i)
  i = random.choice(range(10))
```

A valid output is:

`2,6,0`

`2` is the first random number and `6` is the second random number. `0` is the third random number. It is in red because it is not printed. **Use the `iter` function instead of the `while` loop to do this.**

Try this exercise 50 times and report the average length of the sequence before 0. For example, the length of `2,6,0` is 2; the length of `4,3,9,0` is 3. The average of the two is 2.5.

**Hint: Do not print out all numbers and count the lengths by hand. Create a list `lengths=[]` to store the lengths of sequences and then take the average: `sum(lengths)/50`.**