W9: Intro to Python - Day 3

Fei-Man Hsu, PhD

UCLA QCBio Collaboratory, Fall 2025

This is the notebook for the third day of the Collaboratory Workshop, *Intro to Python*.

Recap

Topics from Day 1 and Day 2:

- Basic data types (str, int, float, boolean, list)
- Decisions and logical expression (operators)
- If statement
- Loops (for, while)
- Built functions (range(), print(), list())
- Reading / Writing files

Variables and basic data types

```
v1 = "Hi there!"
v2 = 300
v3 = 300.0
v4 = 10 < 14
```

If statements

```
snow_outside = True
if snow_outside:
    print("Stay home!")
else:
    print("Go to work!")
```

Loops

While loop

```
i = 1
while i <= 10:
    print(i)
    i += 2
print("Done!")
```

While loop with break

```
i = 1
while True:
    if i > 10:
        break
    print(i)
    i += 2
print("Done!")
```

For loop

```
for i in range(1, 10, 2):
    print(i)
print("Done!")
```

For loop with continue statement

```
i = 1
for i in range(1, 10, 2):
    if i % 3:
        continue
    print(i)
print("Done!")
```

Break, Continue and Pass

- Break: A break statement alters the flow of a loop by terminating it once a specified condition is met.
- Continue: The continue statement is used to skip the remaining code inside a loop for the current iteration only.
- Pass: The pass statement is used when a statement or a condition is required to be present in the program, but we don't want any command or code to execute. It's typically used as a placeholder for future code.

In [1]:
```
for i in range(1, 10, 2): # [1, 3, 5, 7, 9]
    if i % 3:
        continue
    print(i)
```

```
3
9
```

In [4]:
```
for i in range(1, 10, 2): # 1, 3, 5, 7, 9
    if i % 3:
        #print(i)
        break # What would happen if we don't put pass
    print(i)
```

Indentation

- Python indentation refers to adding white space, default 4 spaces, before a statement to a particular `block of code`. In another word, all the statements with the same space to the right, belong to the same code block.
- Python indentation is a way of telling a Python interpreter that the group of statements belongs to a particular block of code.
- A block is a combination of all these statements. Block can be regarded as the grouping of statements `for a specific purpose`.

```
username = "UCLABioInfo2025"
if 10 < len(username) <= 18:      # Block 1
    if username[0].isupper():       # Block 2
        print('Valid')
    else:                           # Block 2
        print('Invalid')
else:                             # Block 1
    print('Invalid')
print("Username validation ends")
```

In [5]:
```
# Nested loop
for i in range(1, 3): #1, 2            # Block 1
    print(i)
    for j in range(1, 4): #1, 2, 3          # Block 2
        print(i*j, end = " ")
    print("")                          # Block 2
print("End of the loop")
```

```
1
1 2 3
2
2 4 6
End of the loop
```

List

- Lists are used to store multiple items in a single variable.
- Lists are one the built-in data types in Python used to store collections of data

In [6]:
```
listOfStrings = ["Mike", "Sam", "Tom"]
print(listOfStrings)
listOfNumbers = [1.0, 3.1, 4.2]
print(listOfNumbers)
listOfMixtype = ["Mike", 1.0, ["Mike", 1.0]]
print(listOfMixtype)
```

```
['Mike', 'Sam', 'Tom']
[1.0, 3.1, 4.2]
['Mike', 1.0, ['Mike', 1.0]]
```

In [7]:
```
# Append() adds an item to a list at the end
listOfStrings.append("Alice")
print(listOfStrings)
```

```
['Mike', 'Sam', 'Tom', 'Alice']
```

In [8]:
```
# Access item with index
print(listOfStrings)
print(listOfStrings[1])
```

```
['Mike', 'Sam', 'Tom', 'Alice']
Sam
```

In [9]:
```
print(listOfMixtype)
print(listOfMixtype[0][2])
```

```
['Mike', 1.0, ['Mike', 1.0]]
k
```

In [10]:
```
print(listOfStrings)
print(listOfStrings[1:3])
```

```
['Mike', 'Sam', 'Tom', 'Alice']
['Sam', 'Tom']
```

In [11]:
```
# List is mutable
print(listOfNumbers)
listOfNumbers[1] = 0.0
print(listOfNumbers)
```

```
[1.0, 3.1, 4.2]
[1.0, 0.0, 4.2]
```

In [13]:
```
# Convert list to string
print(listOfStrings)
listToString = ",".join(listOfStrings)
print(listToString)
print(type(listToString))
```

```
['Mike', 'Sam', 'Tom', 'Alice']
Mike,Sam,Tom,Alice
<class 'str'>
```

In [16]:
```
# Convert string to list
StringToList = list(listToString)
print(StringToList)
print(listToString)
```

```
['M', 'i', 'k', 'e', ',', 'S', 'a', 'm', ',', 'T', 'o', 'm', ',', 'A', 'l', 'i', 'c', 'e']
Mike,Sam,Tom,Alice
```

In [18]:
```
# Try the split() method
print(listToString.split(','))
type(listToString.split(','))
```

```
['Mike', 'Sam', 'Tom', 'Alice']
```

Out[18]:
```
list
```

Outline

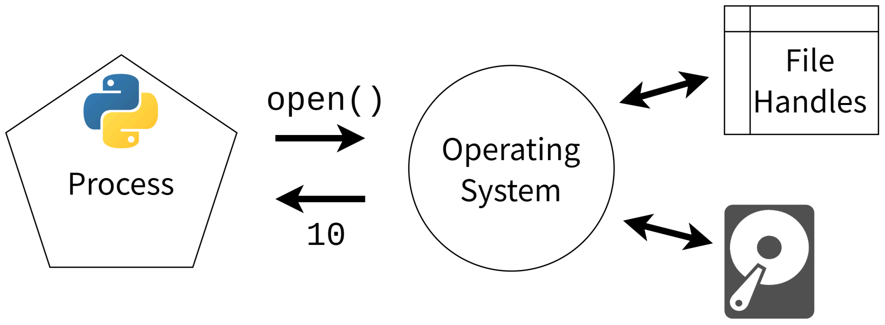
Day 3:

- Reading/Writing files
- Function
- Dictionary

Assignment

- Due by the end of Oct 17

Reading/Writing files



- The `with` statement initiates the `context manager` that opens the file and manages the file resources as long as the context is active (within the indentation block)

In [21]:
```
with open("test.txt", "w") as f:
    f.write("Hello, there!\n")
```

In [22]:
```
with open("test.txt", "r") as f:
    for line in f:
        print(line.strip()) # remove empty spaces before and after strings
```

```
Hello, there!
```

Task 1

- Create a file that store the following sequences in fasta format with tag e.g. Seq1, Seq2 ....
- Read the file and print only the sequences

Example fasta

```
>Seq1
ATGAT
>Seq2
CGATA
>Seq3
TTACT
```

In [24]:
```
Seqs = ['ATGAT', 'CGATA', 'TTACT']
# # Writing file
# with open('Sequence.fasta', 'w') as fout:
#     for i in range(len(Seqs)): # 0, 1, 2
#         fout.write('>Seq' + str(i+1) + '\n')
#         fout.write(Seqs[i] + '\n')
# Reading file
with open('Sequence.fasta') as fin:
    #print(fin)
    for line in fin:
        if line[0] == '>':
            pass
        else:
            print(line.strip())
```

```
ATGAT
CGATA
TTACT
```

- Anaconda users: the Sequence.fasta file will show up in the same directory/folder that your .ipynb opens.
- Colab cloud users: the Sequence.fasta file is stored in the folder icon on the left panel.

In [25]:
```
with open('Sequence.fasta') as fin:
    for line in fin:
        if line[0] == '>':
            continue
        print(line.strip())
```

```
ATGAT
CGATA
TTACT
```

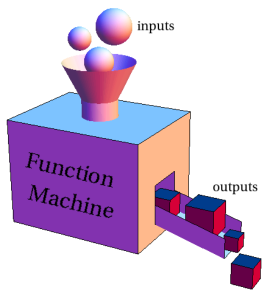
In [26]:
```
with open('Sequence.fasta') as fin:
    for line in fin:
        if not line.startswith('>'):
            print(line.strip())
```

```
ATGAT
CGATA
TTACT
```

Function

Function could serve as a unit in the script

- A **function** is a block of code which only runs when it is called.
- You can pass data, known as **parameters** or **arguments**, into a function. This formally defines "input" of code. - Function can **return** data as a result, or "output".
- After function is defined, it can be called from any part of the script, unlimited times.



Let's define our first function,

```
def NameOfFunction():
    # do something
```

It is a function with no parameters and no return values. No input, no "programmatically retrievable" output.

In [27]:
```
# Define a function say_hello()
def say_hello(): # argument could be empty
    print("Hello, there!")
```

In [28]:
```
say_hello()
say_hello()
say_hello()
```

```
Hello, there!
Hello, there!
Hello, there!
```

Function can accept variables as parameters

```
def say_hello(name):
    print("Hi {}!".format(name))
```

In [29]:
```
def say_hello(name):
    print("Hi {}!".format(name))
```

In [30]:
```
say_hello("Mike")
say_hello("Sam")
say_hello("Tom")
```

```
Hi Mike!
Hi Sam!
Hi Tom!
```

Loop with functions

```
friends = ["Mike", "Sam", "Tom", "Alice"]
for f in friends:
    say_hello(f)
```

In [31]:
```
friends = ["Mike", "Sam", "Tom", "Alice"]
for f in friends:
    say_hello(f)
```

```
Hi Mike!
Hi Sam!
Hi Tom!
Hi Alice!
```

Functions can return values, which formally defines "output" of a code

```
def addition(a, b):
    return a + b
```

In [32]:
```
def addition(a, b):
    return a + b
```

In [34]:
```
a1 = 2
b1 = 3
s = addition(a1, b1)
print("The sum of {} and {} is {}.".format(a1, b1, s))
```

```
The sum of 2 and 3 is 5.
```

- We could break down a complex issue into sub-topics and solve with different functions that build up with hierarchy.
- Make the script organized with blocks
- Functions help in creating **top-down** design for programs.

Example: Say hello to Mike, Sam, Tom and Alice

In [35]:
```
print('Hi, Mike!')
print('Hi, Sam!')
print('Hi, Tom!')
print('Hi, Alice!')
```

```
Hi, Mike!
Hi, Sam!
Hi, Tom!
Hi, Alice!
```

In [36]:
```
def say_hello(name): # parameter is a string
    print("Hi {}!".format(name))
```

In [37]:
```
say_hello('Mike')
say_hello('Sam')
say_hello('Tom')
say_hello('Alice')
```

```
Hi Mike!
Hi Sam!
Hi Tom!
Hi Alice!
```

What if we have 20 friends in the group?

In [38]:
```
def say_hello(name): # parameter is a string
    print("Hi {}!".format(name))
def say_hello_to_all(friends): # parameter is a list
    for f in friends:
        say_hello(f)
```

In [39]:
```
friends = ["Mike", "Sam", "Tom", "Alice"]
say_hello_to_all(friends)
```

```
Hi Mike!
Hi Sam!
Hi Tom!
Hi Alice!
```

- Functions process the repetitive tasks

Example: Celcius to Fahrenheit converter

Input: A single floating-point number, `C`.

Output: Output a single floating-point number, `F`, the temperature in degrees Fahrenheit which is equivalent to `C` degrees Celsius.

$$F = \frac{9}{5}C + 32$$

This time, we will write in a function, named `c_to_f()`.

In [40]:
```python
def c_to_f(C):
    F = (9 / 5) * C + 32
    return F
```

By having this in the form of a function, we can use it in many ways. For interactive input and output, as we did in last two days:

In [41]:
```python
C = float(input("Input degree in C: "))
F = c_to_f(C) # return value of c_to_f
print(F)
```

```
Input degree in C: 32
89.6
```

You can also incorporate this function in other ways, such as reading input from a file, putting output to a file, or inside a weather forecast webpage.

Example

- Write a function test_number that checks if the number is divisible by 3, 5 or 7, but NOT by 15, 21, 35 or 105.
- Use this function to find a sum of integers that pass the test in range from 1 to 100.

In [42]:
```python
def test_number(n):
    return ((n % 3 == 0) or (n % 5 == 0) or (n % 7 == 0)) and not \
    ((n % 15 == 0) or (n % 21 == 0) or (n % 35 == 0) or (n % 105 == 0)) # return a boolean
```

In [43]:
```python
s = 0
for i in range(1, 101):
    if test_number(i): # if True
#         print(i, end = ',')
        s += i
print('\n')
print(s)
```

```


2208
```

Task 2

You might be surprised to know that 2013 is the first year since 1987 with four distinct digits. The years 2014, 2015, 2016, 2017, 2018, 2019 each have four distinct digits. 2012 does not have four distinct digits, since the digit 2 is repeated. Given a year, what is the next year with four distinct digits?

- Input: The input consists of one integer `Y`, representing the starting year. `1000 <= Y < 3000`.
- Output: The output will be a single integer `D`, which is the next year after `Y` with four distinct digits.

Let's solve it using a couple of functions.

In [ ]:
```python
# Write your code here
```

In [44]:
```python
def all_distinct_digits(n):
    l = [] # digit
    i = n
    while i > 0:
        d = i % 10 # remainder
        if d in l:
            return False
        else:
            l.append(d)
        i //= 10 # i = i // 10, quotient from division
    return True
def next_year_with_distinct_digits(n):
    n = n + 1
    while not all_distinct_digits(n):
        n += 1
    return n
```

In [45]:
```
next_year_with_distinct_digits(1987)
```

Out[45]:
```
2013
```

Exercise

- Create a function `print_triangle()` for printing the pattern:
  1
  1 2
  ...
  1 2... n
- Create `print_row()` function and use it inside `print_triangle()`

In [ ]:
```
# Write your code here
```

In [ ]:
```
# Yesterday we did this with nested loops
n = int(input("Input n: "))
for i in range(1, n + 1): # n rows
    for j in range(1, i + 1): # what to print per row
        print(j, end = " ")
    print() # Next line
```

In [46]:
```
def print_row(n):
    for i in range(1, n + 1):
        print(i, end=" ")
    print()

def print_triangle(n):
    for i in range(1, n + 1):
        print_row(i)
```

In [47]:
```
print_triangle(5)
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Exercise

Write a functions that compute factorial number:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

- Loop

In [48]:
```
def factorial_loop(n):
    f = 1
    for i in range(1, n + 1):
        f *= i # f = f * i
        print(f)
    return f
```

In [49]:
```
factorial_loop(5)
```

```
1
2
6
24
120
```

Out[49]:
```
120
```

Exercise

- Write a function `is_leap_year()` that checks if a year is a leap year.

- Using for loop, compute a total number of leap years between AD999 and AD2010
- Modify the program to output all leap year to a file "leap_year.txt".

Hint: To check if a year is a leap year, divide the year by 4. If it is fully divisible by 4, it is a leap year. For example, the year 2016 is divisible 4, so it is a leap year, whereas, 2015 is not. However, Century years like 300, 700, 1900, 2000 need to be divided by 400 to check whether they are leap years or not. Century years not divisible by 400 are not leap years.

In [ ]:
```
# Write your code here
```

In [50]:
```
def is_leap_year(year):
    if year % 4 == 0:
        if year % 100 == 0:
            if year % 400 == 0:
                return True
            else:
                return False
        else:
            return True
    else:
        return False
```

In [51]:
```
cnt = 0
for year in range(999, 2010 + 1):
    if is_leap_year(year): # True
        print(year, end = ",")
        cnt += 1
# print("")
print(cnt)
```

```
1004,1008,1012,1016,1020,1024,1028,1032,1036,1040,1044,1048,1052,1056,1060,1064,1068,1072,1076,1080,1084,108
8,1092,1096,1104,1108,1112,1116,1120,1124,1128,1132,1136,1140,1144,1148,1152,1156,1160,1164,1168,1172,1176,1
180,1184,1188,1192,1196,1200,1204,1208,1212,1216,1220,1224,1228,1232,1236,1240,1244,1248,1252,1256,1260,126
4,1268,1272,1276,1280,1284,1288,1292,1296,1304,1308,1312,1316,1320,1324,1328,1332,1336,1340,1344,1348,1352,1
356,1360,1364,1368,1372,1376,1380,1384,1388,1392,1396,1404,1408,1412,1416,1420,1424,1428,1432,1436,1440,144
4,1448,1452,1456,1460,1464,1468,1472,1476,1480,1484,1488,1492,1496,1504,1508,1512,1516,1520,1524,1528,1532,1
536,1540,1544,1548,1552,1556,1560,1564,1568,1572,1576,1580,1584,1588,1592,1596,1600,1604,1608,1612,1616,162
0,1624,1628,1632,1636,1640,1644,1648,1652,1656,1660,1664,1668,1672,1676,1680,1684,1688,1692,1696,1704,1708,1
712,1716,1720,1724,1728,1732,1736,1740,1744,1748,1752,1756,1760,1764,1768,1772,1776,1780,1784,1788,1792,179
6,1804,1808,1812,1816,1820,1824,1828,1832,1836,1840,1844,1848,1852,1856,1860,1864,1868,1872,1876,1880,1884,1
888,1892,1896,1904,1908,1912,1916,1920,1924,1928,1932,1936,1940,1944,1948,1952,1956,1960,1964,1968,1972,197
6,1980,1984,1988,1992,1996,2000,2004,2008,245
```

In [52]:
```
with open("leap_year.txt", "w") as f:
    for year in range(999, 2010+1):
        if is_leap_year(year):
            f.write(str(year) + "\n")
```

Exercise

Create a function that checks if two lists have at least one element in common.

In [53]:
```
l1 = [1, 2, 3, 4, 5]
l2 = [6, 7, 1, 9, 10]
```

In [54]:
```
def has_something_in_common(l1, l2):
    for i1 in l1:
        for i2 in l2:
            if i1 == i2:
                return True
#             else:
#                 pass
    return False
has_something_in_common(l1, l2) # Boolean
```

Out[54]:
```
True
```

Dictionary

- Dictionaries stores key-value pairs
- Dictionaries like sets use hash-tables for keys and therefore they are very efficient at finding elements

```
d = {"key1": "value1",
     "key2": "value2"
    }
```

In [55]:
```
d = {"name": "Zara",
     "age": 34,
     "position": "software engineer",
     "degree": "BS"
    }
```

In [56]:
```
print(d)
```
```
{'name': 'Zara', 'age': 34, 'position': 'software engineer', 'degree': 'BS'}
```

Dictionaries have `.keys()` method that lists keys in the dictionary:

In [57]:
```
print(d.keys())
```
```
dict_keys(['name', 'age', 'position', 'degree'])
```

Dictionaries also have `.values()` method that lists values in the dictionary.

In [58]:
```
print(d.values())
```
```
dict_values(['Zara', 34, 'software engineer', 'BS'])
```

Also, there is `.items()` method that lists key-value pairs.

In [59]:
```
print(d.items())
```
```
dict_items([('name', 'Zara'), ('age', 34), ('position', 'software engineer'), ('degree', 'BS')])
```

Access the value by its key using square brackets(`[]`):

```
d["key"]
```

In [60]:
```
d["name"]
```
Out[60]:
```
'Zara'
```

In [61]:
```
d["degree"]
```
Out[61]:
```
'BS'
```

- Dictionaries are mutable: you can change the value corresponding to a certain key.

In [62]:
```
d["degree"] = "MS"
d["degree"]
```
Out[62]:
```
'MS'
```

In [63]:
```
d.items()
```
Out[63]:
```
dict_items([('name', 'Zara'), ('age', 34), ('position', 'software engineer'), ('degree', 'MS')])
```

- iterate a dictionary using a for loop:

In [64]:
```
for key in d.keys():
    print("{}: {}".format(key, d[key]))
```
```
name: Zara
age: 34
position: software engineer
degree: MS
```

- iterate a dictionary with `.items()` method:

In [65]:
```
for (k, v) in d.items(): #(key, value)
    print("{}: {}".format(k, v))
```

```
name: Zara
age: 34
position: software engineer
degree: MS
```

Task 3

- Create a dictionary based on the gene expression table below
- Compare the gene expression levels between sampleA and sampleB
- Output a file that list genes overexpressed in sampleA

Example table

Gene	SampleA	SampleB
Gene1	90.3	100.2
Gene2	2.5	20.4
Gene3	20.1	70.3
Gene4	10.7	110.2
Gene5	120.2	50.9

In [66]:
```
GeneExpDict = { 'Gene1': [ 90.3, 100.2],
                'Gene2': [ 2.5 ,  20.4],
                'Gene3': [ 20.1,  70.3],
                'Gene4': [ 10.7, 110.2],
                'Gene5': [120.2,  50.9]
              }
print(GeneExpDict)
```

```
{'Gene1': [90.3, 100.2], 'Gene2': [2.5, 20.4], 'Gene3': [20.1, 70.3], 'Gene4': [10.7, 110.2], 'Gene5': [120.
2, 50.9]}
```

In [67]:
```
print(GeneExpDict.keys())
len(GeneExpDict.items())
```

```
dict_keys(['Gene1', 'Gene2', 'Gene3', 'Gene4', 'Gene5'])
```

Out[67]:
```
5
```

In [68]:
```
print(GeneExpDict.values())
```

```
dict_values([[90.3, 100.2], [2.5, 20.4], [20.1, 70.3], [10.7, 110.2], [120.2, 50.9]])
```

In [69]:
```
print(GeneExpDict.items())
```

```
dict_items([('Gene1', [90.3, 100.2]), ('Gene2', [2.5, 20.4]), ('Gene3', [20.1, 70.3]), ('Gene4', [10.7, 110.
2]), ('Gene5', [120.2, 50.9])])
```

In [72]:
```
overExpA = []
for gene in GeneExpDict.keys():
#     print(gene)
    if GeneExpDict[gene][0] > GeneExpDict[gene][1]:
        overExpA.append(gene)
    else:
        continue
print(overExpA)
```

```
['Gene5']
```

In [71]:
```
with open('SampleA_overexpressed_genes.txt', 'w') as fout:
    for gene in overExpA:
        fout.write(gene + "\n")
```

Task 4

We have created the Sequence.fasta file in Task1,

```
>Seq1
ATGAT
>Seq2
CGATA
>Seq3
TTACT
```

- Read the file into a `dictionary`
  - Sequence name as key
  - Sequence as value
- Write a `function` that return the `reverse complement` sequence when you feed a DNA sequence
- Write the reverse complement sequences into a fasta file "revSequence.fasta" `in the same order` of "Sequence.fasta" but adding "rev_" prefix to each sequence name, e.g. `>Seq1 becomes >rev_Seq1`.

Reverse complement of DNA sequence

- Sequence read in reverse order, ATG becomes GTA.
- Complement |Base | Complement | |:---:|:---:| |A | T | |C | G | |G | C | |T | A |

In [ ]:
```
# Write your code here
```

In [73]:
```
SeqNames, Seqs = [], []
with open("Sequence.fasta") as f_in:
    for line in f_in:
        if line[0] == '>':
            SeqNames.append(line[1:].strip())
        else:
            Seqs.append(line.strip())
print(SeqNames)
print(Seqs)
SeqDict = {}
for i in range(len(SeqNames)):
    SeqDict[SeqNames[i]] = Seqs[i]
print(SeqDict['Seq1'])
```

```
['Seq1', 'Seq2', 'Seq3']
['ATGAT', 'CGATA', 'TTACT']
ATGAT
```

In [74]:
```
def reverseComplement(Seq):
    rcDict = {
        'A' : 'T',
        'C' : 'G',
        'G' : 'C',
        'T' : 'A'
    }
    # Reverse
    Seq = Seq[::-1]
    # Complement
    rSeq = []
    for s in Seq:
        rs = rcDict[s]
        rSeq.append(rs)
    rSeq = ''.join(rSeq)
    return rSeq
```

In [75]:
```
with open("revSequence.fasta", "w") as f_out:
    for i in SeqDict.keys():
        seq = SeqDict[i]
        rseq = reverseComplement(seq)
        f_out.write(">rev_" + i + "\n")
        f_out.write(rseq + "\n")
```

In [76]:
```
with open("revSequence.fasta") as f_in:
    for line in f_in:
        print(line.strip())
```

```
>rev_Seq1
ATCAT
>rev_Seq2
TATCG
>rev_Seq3
AGTAA
```

Running scripts

So far, we have learned how to write python code in Jupyter Notebook. However, for large-scale programs, or if you want to run your program on a shared machine or cluster, Jupyter Notebook is not a good application to run Python code. Now, we will learn how to run code directly in terminal. Jupyter Notebook can also send commands by running cells that starts with a `!`. It is equivalent to typing the command in your terminal.

In [77]:
```
!python --version
```
```
Python 3.11.5
```

In [88]:
```
with open("my_first_script.py", "w") as f_out:
    f_out.write('\"print("This is output from my first script!")') # """ xxx""" comment
```

Let's see the content of the newly-created python script:

In [86]:
```
!cat my_first_script.py
```
```
print("This is output from my first script!")
```

Let's run this python script:

In [80]:
```
!python my_first_script.py
```
```
This is output from my first script!
```

Please scan to evaluate this workshop

Assignment 7

Name	Height (m)	Weight (kg)
Tom	1.78	80
Sara	1.56	52
Jake	1.88	100
Kim	1.66	57
Sean	1.70	60

- Create a dictionary with Name as key and [Height, Weight] as value
- Write a function to calculate the BMI (Weight/(HeightxHeight))
- Append the BMI to the value
- print the new dictionary into the form:

```
Tom     1.78    80     Tom's BMI

Sara    1.56    52     Sara's BMI
```

In [ ]:
```
```

Assignment 8

A DNA sequence is composed of A, C, G and T. Given a user query sequence, write a function to

- Write a function to find potential start codons ('ATG') and stop codons ('TAA', 'TAG', 'TGA')
- Print all potential coding sequences (from start to stop codon the length could be fully divided by 3)

```
GAAGTGGGCAGCCCGCGGCCAACGGACACCCCTCGGGCACCGGCACTTCGGCCTCCACTTCCGGTGTCCG
GCCCGGTCCCCGGGGGCGCTTCTGTGGTGGGGGGTCCCTCAGTGCCTTTCCCCCAAAGCTGTGCATCTCG
ATGGCGGCCTCCAGAGACGGATGTCACCCGTCGGGGCCTGGGCTGGGCTGGCTCATGTTCGAGAGCCCAG
CCTTCCGTGGGGTACTCATTCTGTCAGTTGGGACTGTGTGGTTCTTGCTCCTTGGGAAAGTCCATGTTTT
GGCAGGAACTTTAACCGTGCATTGCAGCGCAGGGCCCGGGAAGCGTGAAGGCTGCATGGTTGGTCTTTTC
```

In [ ]:
```
```

Assignment 9

```
A long time ago in a galaxy far, far away...
```

In the above example, the word far is repeated twice. However, we'd instead like to repeat it exactly $N (1 <= N <= 5)$ times without changing the rest of the sentence at all. There should be a comma right after each occurrence except for the last one.

Given N, can you produce the correct sentence?

```
In [ ]:
```