PYTHON IN ONE SHOT

(ONLY CODE VERSION)

VARIABLES

In [2]:

```
# Variables are dynamically typed
n = 0
print('n = ', n)
```

```
n =  0
```

In [4]:

```
n = "abc"
print('n = ', n)
```

```
n =  abc
```

In [6]:

```
# Multiple Assignments
n,m = 0,"abc"
a,b,c = 0.125, "tmleyncodes", True
print('n = ', n)
print('m = ', m)
print('a = ', a)
print('b = ', b)
print('c = ', c)
```

```
n =  0
m =  abc
a =  0.125
b =  tmleyncodes
c =  True
```

In [9]:

```
# Increments and Decrements
n = 1
print("Before Increment")
print('n = ', n)
n = n+1 # good
print("First Increment")
print('n = ', n)
n += 1 # good
print("Second Increment")
print('n = ', n)
# n++ # bad
print("Final Result: ")
print('n = ', n)
```

```
Before Increment
n =  1
First Increment
n =  2
Second Increment
n =  3
Final Result:
n =  3
```

In [10]:

```
# None is NULL
# None is NULL
a = 4
print('A = ', a)
b = None
print('B = ', b)
```

```
A =  4
B =  None
```

CONDITIONAL STATEMENTS

In [ ]:

```
# If statements don't need parentheses
# or curly braces.
n = 1
if n > 2:
    n -= 1
elif n == 2:
    n *= 2
else:
    n += 2
```

In [ ]:

```
# Parentheses needed for multi-line conditions.
# and = &&
# or  = ||
n, m = 1, 2
if ((n > 2 and
    n != m) or n == m):
    n += 1
```

In [11]:

```
# A simple problem on Age
age = 20
if age>18:
    print("He/She is greater than the age of 18")
else:
    print("He/She is less than the age of 18")
```

```
He/She is greater than the age of 18
```

LOOPS

In [12]:

```
n = 0
print("---While Loop---")
while n<5:
    print(n)
    n+=1
```

```
---While Loop---
0
1
2
3
4
```

In [13]:

```
# For Loop
print("---For Loop---")
for i in range(5):
    print(i)
```

```
---For Loop---
0
1
2
3
4
```

In [14]:

```
# Looping from i = 2 to i = 5
for i in range(2,6):
    print(i)
```

```
2
3
4
5
```

In [15]:

```
# Looping from i = 5 to i = 2
for i in range(5,1,-1):
    print(i)
```

```
5
4
3
2
```

MATHEMATICS

In [16]:

```
# Division is decimal by default
print(5 / 2)
```

```
2.5
```

In [17]:

```
# Double slash rounds down
print(5 // 2)
```

```
2
```

In [18]:

```
# CAREFUL: most languages round towards 0 by default
# So negative numbers will round down
print(-3 // 2)
```

```
-2
```

In [19]:

```
# A workaround for rounding towards zero
# is to use decimal division and then convert to int.
print(int(-3 / 2))
```

```
-1
```

In [20]:

```
# Modding is similar to most languages
print(10 % 3)
```

```
1
```

In [21]:

```
# Except for negative values
print(-10 % 3)
```

```
2
```

In [22]:

```
# To be consistent with other languages modulo
import math
from multiprocessing import heap
print(math.fmod(-10, 3))
```

```
-1.0
```

In [23]:

```
# More math helpers
print(math.floor(3 / 2))
```

```
1
```

In [24]:

```
print(math.ceil(3 / 2))
```

```
2
```

In [25]:

```
print(math.sqrt(2))
```

```
1.4142135623730951
```

In [26]:

```
print(math.pow(2, 3))
```

```
8.0
```

In [ ]:

```
# Max / Min Int
float("inf")
float("-inf")
```

In [27]:

```
# Python numbers are infinite so they never overflow
print(math.pow(2, 200))
```

```
1.6069380442589903e+60
```

In [28]:

```
# But still less than infinity
print(math.pow(2, 200) < float("inf"))
```

```
True
```

ARRAYS

In [29]:

```
# Arrays (called lists in python)
arr = [1, 2, 3]
print(arr)
```

```
[1, 2, 3]
```

In [30]:

```
# Can be used as a stack
arr.append(4)
arr.append(5)
print(arr)
```

```
[1, 2, 3, 4, 5]
```

In [31]:

```
arr.pop()
print(arr)
```

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

In [32]:

```
arr.insert(1, 7)
print(arr)
```

```
[1, 7, 2, 3, 4]
```

In [33]:

```
arr[0] = 0
arr[3] = 0
print(arr)
```

```
[0, 7, 2, 0, 4]
```

In [34]:

```
# Initialize arr of size n with default value of 1
n = 5
arr = [1] * n
print(arr)
```

```
[1, 1, 1, 1, 1]
```

In [35]:

```
print(len(arr))
```

```
5
```

In [36]:

```
# Careful: -1 is not out of bounds, it's the last value
arr = [1, 2, 3]
print(arr[-1])
```

```
3
```

In [37]:

```
# Indexing -2 is the second to last value, etc.
print(arr[-2])
```

```
2
```

In [38]:

```
# Sublists (aka slicing)
arr = [1, 2, 3, 4]
print(arr[1:3])
```

```
[2, 3]
```

In [39]:

```
# Similar to for-loop ranges, last index is non-inclusive
print(arr[0:4])
```

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

In [40]:

```
# But no out of bounds error
print(arr[0:10])
```

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

In [41]:

```
# Unpacking
a, b, c = [1, 2, 3]
print(a, b, c)
```

```
1 2 3
```

In [ ]:

```
# Be careful though, this throws an error
a, b = [1, 2, 3]
```

In [42]:

```
# Looping through arrays
nums = [1, 2, 3]
```

In [43]:

```
# Using index
for i in range(len(nums)):
    print(nums[i])
```

```
1
2
3
```

In [44]:

```
# Without index
for n in nums:
    print(n)
```

```
1
2
3
```

In [45]:

```
# With index and value
for i, n in enumerate(nums):
    print(i, n)
```

```
0 1
1 2
2 3
```

In [46]:

```
# Loop through multiple arrays simultaneously with unpacking
nums1 = [1, 3, 5]
nums2 = [2, 4, 6]
for n1, n2 in zip(nums1, nums2):
    print(n1, n2)
```

```
1 2
3 4
5 6
```

In [47]:

```
# Reverse
nums = [1, 2, 3]
nums.reverse()
print(nums)
```

```
[3, 2, 1]
```

In [48]:

```
# Sorting
arr = [5, 4, 7, 3, 8]
arr.sort()
print(arr)
```

```
[3, 4, 5, 7, 8]
```

In [49]:

```
arr.sort(reverse=True)
print(arr)
```

```
[8, 7, 5, 4, 3]
```

In [50]:

```
arr = ["bob", "alice", "jane", "doe"]
arr.sort()
print(arr)
```

```
['alice', 'bob', 'doe', 'jane']
```

In [51]:

```
# Custom sort (by length of string)
arr.sort(key=lambda x: len(x))
print(arr)
```

```
['bob', 'doe', 'jane', 'alice']
```

In [52]:

```
# List comprehension
arr = [i for i in range(5)]
print(arr)
```

```
[0, 1, 2, 3, 4]
```

In [53]:

```
# 2-D lists
arr = [[0] * 4 for i in range(4)]
print(arr)
print(arr[0][0], arr[3][3])
```

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
0 0
```

In [55]:

```
# This won't work as you expect it to
arr = [[0] * 4] * 4
```

STRINGS

In [56]:

```
# Strings are similar to arrays
s = "abc"
print(s[0:2])
```

```
ab
```

In [ ]:

```
# But they are immutable, this won't work
s[0] = "A"
```

In [57]:

```
# This creates a new string
s += "def"
print(s)
```

```
abcdef
```

In [58]:

```
# Valid numeric strings can be converted
print(int("123") + int("123"))
```

```
246
```

In [59]:

```
# And numbers can be converted to strings
print(str(123) + str(123))
```

```
123123
```

In [60]:

```
# In rare cases you may need the ASCII value of a char
print(ord("a"))
print(ord("b"))
```

```
97
98
```

In [61]:

```
# Combine a list of strings (with an empty string delimitor)
strings = ["ab", "cd", "ef"]
print("".join(strings))
```

```
abcdef
```

QUEUES

In [62]:

```
# Queues (double ended queue)
from collections import deque

queue = deque()
queue.append(1)
queue.append(2)
print(queue)
```

```
deque([1, 2])
```

In [63]:

```
queue.popleft()
print(queue)
```

```
deque([2])
```

In [64]:

```
queue.appendleft(1)
print(queue)
```

```
deque([1, 2])
```

In [65]:

```
queue.pop()
print(queue)
```

```
deque([1])
```

HASHSETS

In [73]:

```
# HashSet
mySet = set()

mySet.add(1)
mySet.add(2)
print(mySet)
```

```
{1, 2}
```

In [67]:

```
print(len(mySet))
```

```
2
```

In [68]:

```
print(1 in mySet)
print(2 in mySet)
print(3 in mySet)
```

```
True
True
False
```

In [74]:

```
mySet.remove(2)
print(2 in mySet)
```

```
False
```

In [75]:

```
# list to set
print(set([1, 2, 3]))
```

```
{1, 2, 3}
```

In [76]:

```
# Set comprehension
mySet = { i for i in range(5) }
print(mySet)
```

```
{0, 1, 2, 3, 4}
```

HASHMAPS

In [78]:

```
# HashMap (aka dict)
myMap = {}
myMap["alice"] = 88
myMap["bob"] = 77
print(myMap)
```

```
{'alice': 88, 'bob': 77}
```

In [79]:

```
print(len(myMap))
```

```
2
```

In [80]:

```
myMap["alice"] = 80
print(myMap["alice"])
```

```
80
```

In [81]:

```
print("alice" in myMap)
```

```
True
```

In [82]:

```
myMap.pop("alice")
print("alice" in myMap)
```

```
False
```

In [83]:

```
myMap = { "alice": 90, "bob": 70 }
print(myMap)
```

```
{'alice': 90, 'bob': 70}
```

In [84]:

```
# Dict comprehension
myMap = { i: 2*i for i in range(3) }
```

```
print(myMap)
```

```
{0: 0, 1: 2, 2: 4}
```

In [85]:

```
# Looping through maps
myMap = { "alice": 90, "bob": 70 }
for key in myMap:
    print(key, myMap[key])
```

```
alice 90
bob 70
```

In [86]:

```
for val in myMap.values():
    print(val)
```

```
90
70
```

In [87]:

```
for key, val in myMap.items():
    print(key, val)
```

```
alice 90
bob 70
```

TUPLES

In [88]:

```
# Tuples are like arrays but immutable
tup = (1, 2, 3)
print(tup)
```

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

In [89]:

```
print(tup[0])
print(tup[-1])
```

```
1
3
```

In [ ]:

```
# Can't modify, this won't work
tup[0] = 0
```

In [91]:

```
# Can be used as key for hash map/set
myMap = { (1,2): 3 }
print(myMap[(1,2)])
```

```
3
```

In [92]:

```
mySet = set()
mySet.add((1, 2))
print((1, 2) in mySet)
```

```
True
```

In [ ]:

```
# Lists can't be keys
myMap[[3, 4]] = 5
```

HEAPS

By Default in Python the heap which is built is the `min-heap`.

In [93]:

```
import heapq

# under the hood are arrays
minHeap = []
heapq.heappush(minHeap, 3)
heapq.heappush(minHeap, 2)
heapq.heappush(minHeap, 4)

# Min is always at index 0
print(minHeap[0])
```

```
2
```

In [94]:

```
while len(minHeap):
    print(heapq.heappop(minHeap))
```

```
2
3
4
```

In [95]:

```
# No max heaps by default, work around is
# to use min heap and multiply by -1 when push & pop.
maxHeap = []
heapq.heappush(maxHeap, -3)
heapq.heappush(maxHeap, -2)
heapq.heappush(maxHeap, -4)

# Max is always at index 0
print(-1 * maxHeap[0])
```

```
4
```

In [96]:

```
while len(maxHeap):
    print(-1 * heapq.heappop(maxHeap))
```

```
4
3
2
```

In [97]:

```
# Build heap from initial values
arr = [2, 1, 8, 4, 5]
heapq.heapify(arr)
while arr:
    print(heapq.heappop(arr))
```

```
1
2
4
5
8
```

FUNCTIONS

In [98]:

```
def myFunc(n, m):
    return n * m

print(myFunc(3, 4))
```

```
12
```

In [99]:

```
# Nested functions have access to outer variables
def outer(a, b):
    c = "c"

    def inner():
        return a + b + c
    return inner()

print(outer("a", "b"))
```

```
abc
```

In [101]:

```
# Can modify objects but not reassign
# unless using nonlocal keyword
def double(arr, val):
    def helper():
        # Modifying array works
        for i, n in enumerate(arr):
            arr[i] *= 2

        # will only modify val in the helper scope
        # val *= 2

        # this will modify val outside helper scope
        nonlocal val
        val *= 2
    helper()
    print(arr, val)
nums = [1, 2]
val = 3
double(nums, val)
```

```
[2, 4] 6
```

CLASSES

In [102]:

```
class MyClass:
    # Constructor
    def __init__(self, nums):
        # Create member variables
        self.nums = nums
        self.size = len(nums)

    # self key word required as param
    def getLength(self):
        return self.size

    def getDoubleLength(self):
        return 2 * self.getLength()

myObj = MyClass([1, 2, 3])
print(myObj.getLength())
```

```
3
```

In [103]:

```
print(myObj.getDoubleLength())
```

```
6
```