

# **LE/EECS 1015**

## **(Section D)**

# **Week 10: Collections II**

**Shogo Toyonaga**  
**York University**

**Lassonde School of Engineering**

# This Week...

## 1. Data Structures (Brief Review)

- Tuple
- List
- Set
- Dictionary

## 2. Function Arguments with Collections

## 3. Iterables (Constructors & Loops)

## 4. Coding Style

## 5. Collection Memory Model

# Goals of Lab 8

- 1. Writing modular code with collections**
- 2. Review of Loops (For, While)**
- 3. Debugging**
- 4. Dictionary Methods & Tuple Unpacking**

# Lab 8 – What You Do....

| Task   | Points |
|--|--------|
| Follow the Steps (Merge Lists)                 | 30     |
| Debugging (Contains Duplicates)                | 30     |
| Implementation (Majority Element)              | 10     |
| Implementation (Update & Return Subject Grade) | 20     |
| Implementation (Invert Dictionary)             | 10     |

# Lab 8 – Useful Resources

- [for..in loops in Python for iterating through collections of values or indices within a range](#)
- [Python: Deep Copy vs Shallow Copy](#)
- [Learn Python LIST COMPREHENSIONS in 10 minutes! \(BroCode\)](#)
- [Python Documentation: Shallow and Deep Copying](#)

# Functions (ft. Packing & Unpacking)

## Using a Tuple...

- A function parameter beginning with, “`*`” packs a sequence of values into a tuple.
- Likewise, you can use, “`*`” following an iterable in a function call to unpack the sequence.

## Using a Dictionary...

- A function parameter beginning with, “`**`” packs a sequence of keyword arguments as a dictionary.
- Likewise, you can use, “`**`” following a dictionary variable in a function call to unpack the corresponding keyword arguments.

# A Review of Iterables

- Given an iterable, you can use the built-in constructors to convert between different data structures.
  - *String* (`str(...)`) → `""`
  - *Tuple* (`tuple(...)`) → `()`
  - *List* (`list(...)`) → `[]`
  - *Set* (`set(...)`)
  - *Dict* (`dict(...)`) → `{}`

# Looping

```
# Generate an array of 10 integers between [-100, 100]
simple_sequence = [randint(-100, 100) for _ in range(10)]

# Looping Style 1: By Element
print('===== Style 1: Loop by Elements =====')
for num in simple_sequence:
    print(num, end=' ')
print('')

# Looping Style 2: By Index
print('===== Style 2: Loop by Index =====')
for i in range(len(simple_sequence)):
    print(f'simple_sequence[{i}] = {simple_sequence[i]}')

# List Comprehension
print('===== List Comprehension (Raise to Power of 2) =====')
squared_sequence = [n ** 2 for n in simple_sequence]
print(squared_sequence)
```

# Looping

```
===== Style 1: Loop by Elements =====
-75 19 0 -40 5 78 -41 -91 98 11
===== Style 2: Loop by Index =====
simple_sequence[0] = -75
simple_sequence[1] = 19
simple_sequence[2] = 0
simple_sequence[3] = -40
simple_sequence[4] = 5
simple_sequence[5] = 78
simple_sequence[6] = -41
simple_sequence[7] = -91
simple_sequence[8] = 98
simple_sequence[9] = 11
===== List Comprehension (Raise to Power of 2) =====
[5625, 361, 0, 1600, 25, 6084, 1681, 8281, 9604, 121]
```

# Looping

```
cad_currency_exchange = {  
    'US Dollar': 0.71,  
    'Euro': 0.62,  
    'Pound': 0.54,  
    'Rupee': 63.29,  
    'Yen': 109.83  
}  
  
for conv,scaler in cad_currency_exchange.items():  
    print(f'1 CAD scales to {scaler} {conv}')  
✓ 0.0s  
  
1 CAD scales to 0.71 US Dollar  
1 CAD scales to 0.62 Euro  
1 CAD scales to 0.54 Pound  
1 CAD scales to 63.29 Rupee  
1 CAD scales to 109.83 Yen
```

# Looping

**(True / False): All collections can be iterated through by using indexes (e.g., *nums[i]*).**

# Coding Style

- 1. In general, the elements of a collection should all share the same data type.**
- 2. Python Comprehension offers a clean, concise, and simple technique for creating new sequences from existing ones. You can also apply data transformations in real-time!**

# Coding Style & Collection Memory Model

## 1. Pass by Value (Copy)

- Copies the **value** of an argument to a non-pointer or non-reference.
- If the original value or copy changes, **one does not affect the other**.

## 2. Pass by Reference (Assign)

- Passes the **reference (memory address)** of an argument to a **pointer** or new variable.
- Changes to the variable affect the original reference and vice-versa.
- When you pass a mutable collection data type in a function, the data will be updated globally.

# Collection Memory Model

```
# Assignment
a = [1000, 2000, 3000, 4000]
# Assignment (Pointer)
b = a
# Shallow Copy
c = a.copy()

# Status (Lists)
print('a (original list):', id(a), a)
print('b (assigned list):', id(b), a)
print('c (shallow copy of a):', id(c), c)

# Status (Elements)
for i in range(len(a)):
    print(f'\nid(a[{i}]) == id(b[{i}]) == id(c[{i}]): {id(a[i]) == id(b[i]) == id(c[i])}')

print('Change a[0] = 0')
a[0] = 0

# ===== What do you think print(a,b,c) will look like now?
```

# Collection Memory Model

```
# Extension
m1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
m2 = m1
m3 = m1.copy()

# Status (Lists)
print('m1 (original list):', id(m1), m1)
print('m2 (assigned list):', id(m2), m2)
print('m3 (shallow copy of m1):', id(m3), m3)

# Status (Elements)
for i in range(len(m1)):
    print(f'\tid(m1[{i}]) == id(m2[{i}]) == id(m3[{i}]): {id(m1[i]) == id(m2[i]) == id(m3[i])}')

print('Change m1[0][0] = 0')
m1[0][0] = 0

# ===== What do you think print(m1,m2,m3) will look like now?
```

# Collection Memory Model

```
# Extension (A Peek Into DeepCopy)
m1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
m2 = m1
m3 = m1.copy()
m4 = deepcopy(m1)

# Status (Lists)
print('m1 (original list):', id(m1), m1)
print('m2 (assigned list):', id(m2), m2)
print('m3 (shallow copy of m1):', id(m3), m3)
print('m4 (deep copy of m1):', id(m4), m4)

# Status (Elements)
for i in range(len(m1)):
    print(f'\tid(m1[{i}]) == id(m2[{i}]) == id(m3[{i}]) == id(m4[{i}]): {id(m1[i]) == id(m2[i]) == id(m3[i]) == id(m4[i])}')

print('Change m1[0][0] = "A"')
m1[0][0] = 'A'
print('Change m2[1][0] = "B"')
m2[1][0] = 'B'
print('Change m3[2][0] = "C"')
m3[2][0] = 'C'

# ===== What do you think print(m1,m2,m3) will look like now?
```

# Collection Memory Model

- Shallow Copy
  - “Constructs a **new compound object** and then (to the extent possible) inserts **references** into it to the objects found in the original.”
- Deep Copy
  - “Constructs a **new compound object** and then, **recursively**, inserts **copies** into it of the objects found in the original.”

# Thank You!

Shogo Toyonaga  
Lassonde School of  
Engineering

